Research and Development Technical Report

ECOM-0084-F2

A PROGRAM TO TAKE THE DERIVATIVE OF REGULAR EXPRESSIONS

TECHNICAL REPORT EER 16-7

By

William Hartmann, M.S.

FEBRUARY 1972

ECOM

UNITED STATES ARMY ELECTRONICS COMMAND - FORT MONMOUTH, N.J.

CONTRACT DAAB07-68-C-0084

Avionics Research Group
Department of Electrical Engineering
Ohio University
Athens, Ohio 45701

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position, unless so designated by other authorized documents.

The citation of trade names and names of manufacturers in this report is not to be construed as official Government indorsement or approval of commercial products or services referenced herein.

Disposition

Destroy this report when it is no longer needed. Do not return it to the originator.

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author) Avionics Research Group Department of Electrical Engineering Ohio University, Athens, Ohio 45701	2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED 2b. GROUP

3. REPORT TITLE

A Program to Take the Derivative of Regular Expressions

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)

5. AUTHOR(S) (First name, middle initial, last name)

William Hartmann

6. REPORT DATE February 1972	7a. TOTAL NO. OF PAGES 112	7b. NO. OF REFS 9
8a. CONTRACT OR GRANT NO. DAAB07-68-C-0084 b. PROJECT NO. c. d.	9a. ORIGINATOR'S REPORT NUMBER(S) Technical Report EER 16-7 9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report) ECOM 0084-F2	

10. DISTRIBUTION STATEMENT

Approved for Public Release; Distribution Unlimited

11. SUPPLEMENTARY NOTES Two Reports required under Contract ECOM-0084-F1 and F2	12. SPONSORING MILITARY ACTIVITY US Army Electronics Command ATTN: AMSEL-VL-N Fort Monmouth, New Jersey 07703

13. ABSTRACT

In analyzing a finite state, sequential machine the designer will often use a flowgraph or flow-chart to describe the internal characteristics of the machine. From these characteristics he can obtain a model representing the external performance of the machine. By the external performance of the machine we are referring to the input, output characteristics; i. e., for a given set of input signals what is the output?

For simplicity ~~we will only~~ consider machines with two outputs, a 1 or a 0. Thus, the inputs may be divided into two classes, those which produce a 1 output (accepted or desired inputs) and the remainder which produce a 0 output (the rejected inputs). The regular expression provides a formal method for representing all of the possible inputs which are accepted.

While the regular expression is a powerful tool its use has been limited by the overwhelming amount of work needed to obtain the sequential machine. This paper describes a program which was written to find the derivatives of the regular expression. The program was written for use with the SCC 650 digital computer of the Electrical Engineering Department, Ohio University. This program is called REXPRO for Regular EXpression PROcessor.

UNCLASSIFIED
Security Classification

Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Computer Science Programming (Program for Derivative of Regular Expressions)						

UNCLASSIFIED

TR ECOM-0084-F2
February 1972

Reports Control Symbol
OSD-1366

A PROGRAM TO TAKE THE DERIVATIVE OF REGULAR EXPRESSIONS

TECHNICAL REPORT EER 16-7

Contract No. DAAB07-68-C-0084

Prepared by

William Hartmann, M. S.

Avionics Research Group
Department of Electrical Engineering
Ohio University
Athens, Ohio 45701

For

U. S. ARMY ELECTRONICS COMMAND, FORT MONMOUTH, N. J.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

In analyzing a finite state, sequential machine the designer will often use a flow-graph or flow-chart to describe the internal characteristics of the machine. From these characteristics he can obtain a model representing the external performance of the machine. By the external performance of the machine we are referring to the input, output characteristics; i.e., for a given set of input signals what is the output?

For simplicity we will only consider machines with two outputs, a 1 or a 0. Thus, the inputs may be divided into two classes, those which produce a 1 output (accepted or desired inputs) and the remainder which produce a 0 output (the rejected inputs). The regular expression provides a formal method for representing all of the possible inputs which are accepted.

The designer is often faced with the inverse problem of trying to design a machine to accept the set of desired inputs and to reject all others. If the desired inputs are represented as a regular expression the designer can obtain the internal characteristics, of the machine, from the derivative of the regular expression.

While the regular expression is a powerful tool its use has been limited by the overwhelming amount of work needed to obtain the sequential machine. This paper describes a program which was written to find the derivatives of the regular expression. The program was written for use with the SCC 650 digital computer of the Electrical Engineering Department, Ohio University. This program is called

REXPRO for Regular EXpression PROcessor. While REXPRO will not perform all of the operations needed to design a machine it does do the "dirty work" associated with the design.

Chapter II defines the regular expression and its derivative for the reader who is not familiar with this technique. Several examples using the regular expression are presented in this chapter. Chapter III introduces the concepts of string processing; a method by which the computer is used to operate on non-numerical data such as the regular expression. Chapter IV introduces the methods used in REXPRO for identifying the form of the regular expression and which rule should be used to form the derivative. Chapter V presents an outline of REXPRO and describes its use.

CHAPTER II

THE REGULAR EXPRESSION

The finite state, sequential machine is a machine with a finite number of states that the machine can be in, and the machine can be in one and only one state at any given time. An output is associated with each of the states. This is the description of the "Moore" machine. This type of machine will be used exclusively in this paper, but the results can be modified to include other forms.

The inputs to the machine consist of a set of characters, or symbols, which are called literals. When a given input is seen the machine changes from the present state to some new state. The new state is determined by the present state and the present input. This is shown in Figure II.1.

In this example the machine is initially in state q_1 and has an output of 0. If the input symbol is an 'A' then the machine will go to state q_2 and produce a 1 output. If a 'B'[1] is seen the machine will go to state q_3 and produce a zero output.

The machine shown in Figure II.1 may also be described by the sequence, or string, of characters which will take the machine from a starting state to a state which will produce a 1. One of the sequences is the string 'A'. The symbol 'B' will take the machine to state q_3 and then a second 'B' will take the

[1] The literals 'A' and 'B' are not to be confused with the letter of the alphabet. They are the names given to two of the possible input symbols; e.g., they may represent the polarity of a voltage.

machine from state q_3 to state q_2 so that the string 'BB' is another sequence which will produce a 1 output. Continuing we find the strings: 'A', 'BB', 'BAA', 'BABBA', 'BABBBA', etc.

The regular set consists of the set of <u>all</u> of the possible input sequences which will produce a 1 output. For this example we obtain,

$$S = \{A, BB, BAA, BABA, BABBA, BABBBA, \text{etc.}\}$$

The regular expression is a finite function which represents all of the sequences contained in the regular set. The regular expression is formed by recursively using the Boolean operators (AND, OR, and NOT), the concatenation operator, and the star operator. For this example the regular expression,

$$R = A+BB+BAB^*A$$

is obtained.[2] The star operator is defined as,

$$A^* = \lambda+A+AA+AAA+\ldots\ldots+$$

where the lambda (λ) represents the null string, or string of zero length. Another character which may appear in the regular expression is "Ø", representing the null set.

---

[2] Throughout this paper the AND operator is denoted by the ".", the OR by the "+", the concatenation operator by juxtaposition, and the NOT operator by brackets; e.g., [A] = ~(A).

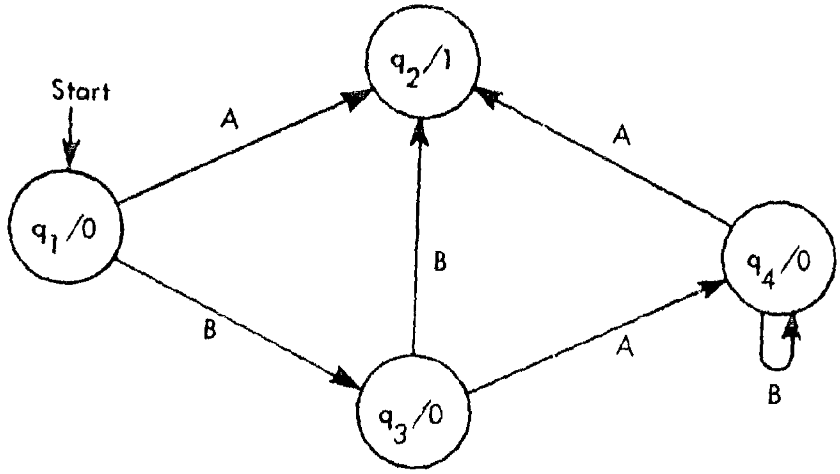
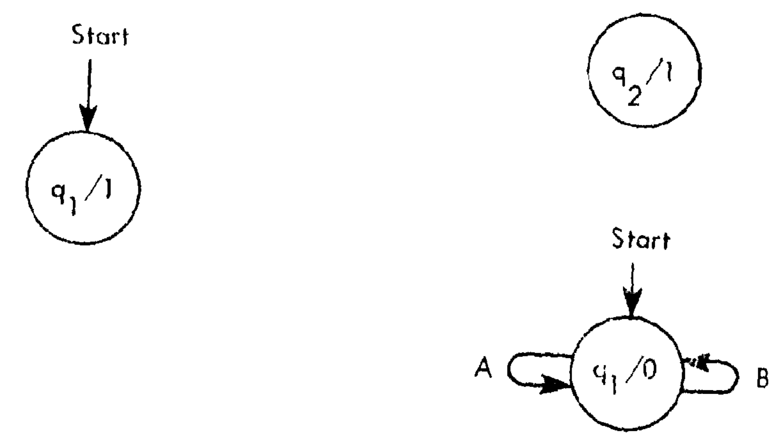


Figure II.1 An Example of a Finite State Machine



(a) $R = \lambda$

(b) $R = \emptyset$

Figure II.2 The Use of λ and $\emptyset$

Figure II.2a shows the meaning of the lambda. In this example the initial state already has the desired output (1) so a sequence of zero length is needed to go from the initial state to the desired state. In Figure II.2b there is no way to go from the initial state to a state which will produce a 1 output. Thus, the regular expression, $R = \emptyset$.

The Derivative Concept

For the rest of this paper we are going to be concerned with the inverse problem; or the design problem. In this class of problems the designer knows the sequence of characters that is required to produce a 1 output.

To design a machine to accept this sequence the concept of the derivative of a regular expression is introduced. The rules for forming the derivative are presented below. The derivation of these rules can be found in [2] .

II.1 $\quad D_a a = \lambda$

II.2 $\quad D_a b = \emptyset$, for $b = \emptyset$, $b = \lambda$, b a literal $\neq a$

II.3 $\quad D_a(X^*) = D_a(X)X^*$

II.4 $\quad D_a(XY) = D_a(X)Y + \eta(X)\, D_a(Y)$

II.5 $\quad D_a(f(X,Y)) = f(D_a(X), D_a(Y))$

II.6 $\quad D_{ab}(X) = D_b(D_a(X))$

Where a and b are literals, X and Y are regular expressions or the result of taking the derivative, and f is any Boolean function. The eta function is defined as,

II.7 $\quad \eta(X) = \begin{cases} \lambda \text{ if } \lambda \in X \\ \emptyset \text{ if } \lambda \notin X \end{cases}$

II.8 $\quad \eta(XY) = \eta(X) \cdot \eta(Y)$

II.9 $\quad \eta(f(X,Y)) = f(\eta(X), \eta(Y))$

It can be shown that any derivative can be found by repeatedly applying the above rules.

To simplify the expression that is formed by using the derivative operation the following identities are presented. Additional identities can be found in [3].

II.10 $\quad \emptyset X = X\emptyset = \emptyset$

II.11 $\quad \emptyset + X = X + \emptyset = X$

II.12 $\quad \lambda X = X\lambda = X$

II.13 $\quad \emptyset \cdot X = X \cdot \emptyset = \emptyset$

The process of obtaining the machine from the regular expression is a simple, but tedious operation. First, associate the regular expression with an initial state (q_1). Second, take the derivative of the regular expression with respect to each of the literals and assign a state to each one of these derivatives. Each of these new states are connected to the initial state via a line directed from the initial state to the new state. Each of these lines is given a value corresponding to the literal which produced the derivative that was assigned to that state. Third, take the derivative of each of the expressions found in step two. This process is repeated until no new expressions are formed. Fourth, the expression assigned to each of the states of the machine is tested to see if it contains lambda (this is the same as the eta function mentioned above). Then a 1 output is assigned to each state for which its corresponding expression contains lambda. A 0 output is assigned to all other states.

For the example shown in Figure II.1 the following expressions are obtained[3].

II.14 R = A + BB + BAB*A

II.15 D/A/ (R) = λ, contains lambda

II.16 D/B/ (R) = B + AB*A

II.17 D/BA/ (R) = B*A

II.18 D/BB/ (R) = λ, contains lambda

II.19 D/AA/ (R) = λ, contains lambda

II.20 D/BAB/ (R) = B*A = D/BA/ (R)

In step 1 the regular expression is associated with the initial state q_1. In the second step the derivative with respect to A was formed. As this is a new expression it was associated with state q_2. Also this expression contains lambda so this state has an output of 1. The literal A was used to generate this expression so a transition from q_1 to q_2 occurs for the input A. Repeating the process state q_3 is generated and it is connected to q_1 by the symbol B.

In step four the derivative, D/BA/ (R) = B*A, is formed. As this is a new expression it is assigned to state q_4. By rule II.6 this derivative can be formed by, D/BA/ (R) = D/A/ (D/B/ (R)) = D/A/ (B + AB*A) = B*A. As the expression assigned to state q_4 is formed by taking the derivative (with respect to A) of the expression assigned to state q_3 we conclude that a transition occurs from q_3 to q_4 for the input symbol A.

---

[3] In keeping with the notation used with REXPRO the slash will be used to indicate the name of the derivative instead of using subscripts; i.e., D/A/ (R) = D_a (R).

The process is continued in steps 5, 6, and 7. The expressions formed in each of these steps are the same as ones formed in the previous steps. In this case the expression is assigned to the same state as it was assigned to in the earlier step. For example, the expression formed in step 5 is assigned to state q_4 and a line is drawn to connect q_4 to q_4 to indicate the transition for the input B.

The expression assigned to state q_2 contains lambda so this state is assigned an output of 1. None of the expressions associated with the remaining states contain lambda, so they are assigned output of 0.

Via this set of operations it has been possible to regenerate the flow-graph of Figure II.1. For the interested reader a more detailed discussion of the derivative of the regular expression is given in [1] and [2]. In the following section several practical, but simplified, problems will be studied. In these problems the machine is not specified beforehand, but the regular expression is used to design a machine to meet the given problem.

Examples Using the Regular Expression

Problem 1. In a small, general purpose computer, such as the SCC 650, the instruction repertoire consists of three classes, the nonmemory instructions, the memory reference instructions, and the indirect memory instructions. The nonmemory instructions are those which do not refer to data stored in the memory, while the memory reference instructions decode part of the instruction to find the address where the data is stored. The indirect instructions first finds the data, as for the memory reference instructions, and then interprets this data as the address where the data is stored in the memory.

A sub-section of the control system is needed to control whether the memory address is to be obtained from the PC register or the LC register. The LC, location counter, register contains the address of the data[4] and the PC, program counter, gives the address for the next instruction. As part of the computer there is an instruction decoder which will indicate which class the instruction is in. The classes will be encoded as N, M, or I for the nonmemory, the memory, and the indirect instructions, respectively. There is also a memory control system which indicates when the memory has finished its read (and write) operation. This completion will be assigned the character F.

Thus the machine which will control the use of the registers has input sequences of N, or MF, or IFF. Or in terms of the regular expression, R = N + MF + IFF. This is not entirely correct as the machine must be able to accept an arbitrarily large number of these sequences. This problem is solved by the use of the star operator.

Thus the expression,

$$R = (N + MF + IFF)^*$$

is obtained. In Figure II.3 the derivatives of this expression are listed as they were obtained from REXPRO[5].

---

4 The problem of how, and when the registers are set will not be considered here.

5 The computer uses the symbol, \ , to indicate the symbol λ and the ,-, is used to indicate the symbol Ø.

```
R-EXP PRØGRAM--READY
R=(N+MF+IFF)*@
EXECUTE

CHECK CØPY..
D((N+MF+IFF)*)
D/N/

D/N/=
(N+MF+IFF)*

CØNTAINS \
D/M/

D/M/=
F(N+MF+IFF)*
D/F/

D/F/=
-
D/I/

D/I/=
FF(N+MF+IFF)*
D/MN/

D/MN/=
-
D/MM/

D/MM/=
-
D/MF/

D/MF/=
(N+MF+IFF)*

CØNTAINS \
D/MI/

D/MI/=
-
```

Figure II.3 Derivatives of R = (N + NF + IFF)*

```
D/IN/

D/IN/=
-
D/IM/

D/IM/=
-
D/IF/

D/IF/=
F(N+MF+IFF)*
D/II/

D/II/=
-
D/IFN/

D/IFN/=
-
D/IFM/

D/IFM/=
-
D/IFF/

D/IFF/=
(N+MF+IFF)*

CØNTAINS \
D/IFI/

D/IFI/=
-
```

Figure II.3 Continued

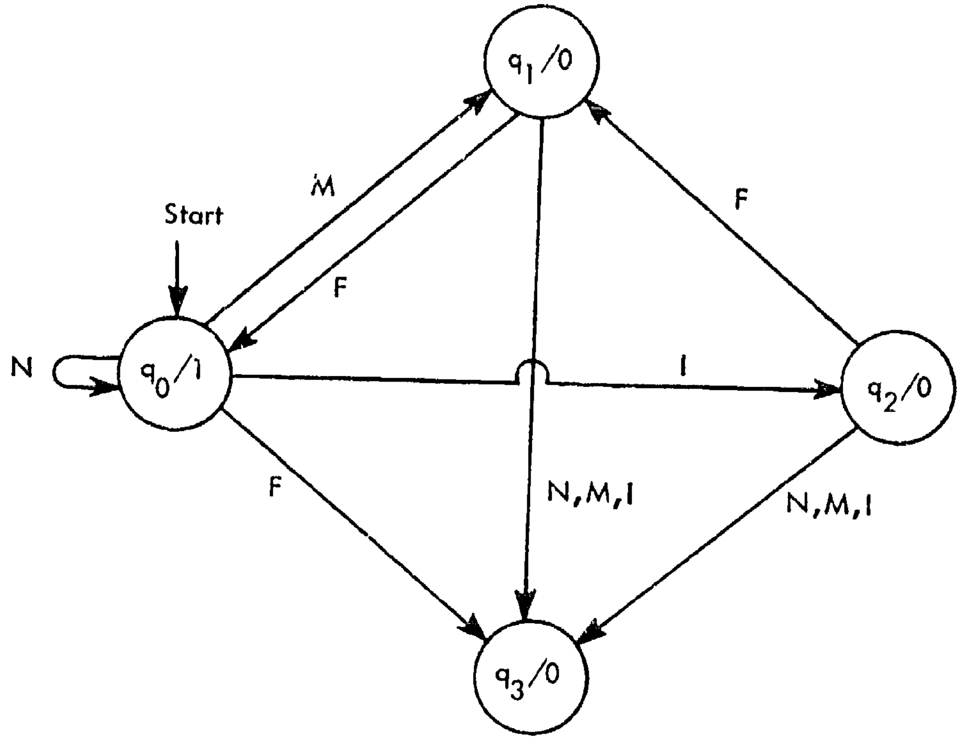


Figure II.4 Flow-graph for Memory Control System

Figure II.4 shows the flow-graph that was obtained from the derivatives. The state with a 1 output is the one for which the PC register is used. The LC register will be used with the rest of the states. State q_3 is entered when illegal sequences of symbols are found. This may be used for error checking.

Problem 2. The reader should be aware of the fact that the finite state, sequential machine is not restricted to hardware designs, but that the execution of a computer program can be studied as a sequential machine.

In computing the eta function it is necessary to test a sub-sequence of the regular expression to see if it contains lambda [6]. The possible input characters in this sub-sequence are the literals, which will be denoted by the symbol A; the phi, denoted by P; the lambda, denoted by L; the star operator, denoted by S; and any other operators, denoted by 0.

For the sequence to contain lambda it must contain only terms of the form AS, L, LS, or PS and an arbitrarily large number, but at least one, of these terms. Thus, the expression,

$$(AS + L + LS + PS)\ (AS + L + LS + PS)^*$$

is generated.

An additional requirement is that the sequence be terminated by an operator. Thus,

$$(AS + L + LS + PS)\ (AS + L + LS + PS)^*0$$

---

6 This is part of the function of the subroutine ØBETAT described in Chapter IV and Appendix D.

To this expression the NULL character, denoted by N, is added for completeness. The NULL character has no intrinsic value, but is a blank, or spacing, character which is placed in the sequence to fill in any unused locations in the sequence. As an arbitrary number of NULL's, including zero, may appear in the sequence the NULL will be placed in the sequence as N*. These NULL's can be placed in the sequence between any and all of the other characters, thus the regular expression given below is obtained.

$$R = (N^*AN^*SN^* + N^*LN^* + N^*LN^*SN^* + N^*PN^*SN^*)$$

$$(N^*AN^*SN^* + N^*LN^*+N^*LN^*SN^* + N^*PN^*SN^*)^*N^*0$$

An equivalent, but simpler, form for the regular expression is,

$$R = N^*Z(N^*Z)^*N^*0, \text{ where}$$

$$Z = (AN^*S + L + LN^*S + PN^*S)$$

Figure II.5 shows the derivatives which were calculated for this problem. Figure II.6 shows the resulting flow-graph. After a sequence has been applied to this machine the final state will either be q_5 or q_7. State q_5 has a one output indicating that the sequence contained lambda; state q_7 produces a zero output for those sequences which do not contain lambda.

It is now a simple matter for the programmer to translate the flow-graph in Figure II.6 to a program which will calculate the eta function. Each of the states, except the terminal states q_7 and q_5, is translated into a subprogram which

```
K-EXP PRØGRAM--READY
R=N*Z(N*Z)*N*Ø@
Z=AN*S+L+LN*S+PN*S@
EXECUTE

CHECK CØPY..
D(N*(AN*S+L+LN*S+PN*S)(N*(AN*S+L+LN*S+PN*S))*N*Ø)
D/N/

D/N/=
(N)*(AN*S+L+LN*S+PN*S)(N*(AN*S+L+LN*S+PN*S))*N*Ø
D/A/

D/A/=
(N)*S(N*(AN*S+L+LN*S+PN*S))*N*Ø
D/S/

D/S/=
-
D/L/

D/L/=
(\+(N)*S)(N*(AN*S+L+LN*S+PN*S))*N*Ø
D/P/

D/P/=
(N)*S(N*(AN*S+L+LN*S+PN*S))*N*Ø
D/Ø/

D/Ø/=
-
D/AN/

D/AN/=
(N)*S(N*(AN*S+L+LN*S+PN*S))*N*Ø
D/AS/

D/AS/=
(N*(AN*S+L+LN*S+PN*S))*N*Ø
D/AA/

D/AA/=
-
```

Figure II.5 Derivatives of R = N*Z (N*Z)*N*0

```
D/AL/

D/AL/=
-
D/AP/

D/AP/=
-
D/AØ/

D/AØ/=
-
D/ASN/

D/ASN/=
((N)*(AN*S+L+LN*S+PN*S)(N*(AN*S+L+LN*S+PN*S))*N*Ø+(N)*Ø)

D/ASA/

D/ASA/=
(N)*S(N*(AN*S+L+LN*S+PN*S))*N*Ø
D/ASS/

D/ASS/=
-
D/ASL/

D/ASL/=
(\+(N)*S)(N*(AN*S+L+LN*S+PN*S))*N*Ø
D/ASP/

D/ASP/=
(N)*S(N*(AN*S+L+LN*S+PN*S))*N*Ø
D/ASØ/

D/ASØ/=
\

CØNTAINS \
D/ASNN/

D/ASNN/=
((N)*(AN*S+L+LN*S+PN*S)(N*(AN*S+L+LN*S+PN*S))*N*Ø+(N)*Ø)
```

Figure II.5 Continued

```
D/ASNA/

D/ASNA/=
(N)*S(N*(AN*S+L+LN*S+PN*S))*N*Ø
D/ASNS/

D/ASNS/=
-
D/ASNL/

D/ASNL/=
(\+(N)*S)(N*(AN*S+L+LN*S+PN*S))*N*Ø
D/ASNP/

D/ASNP/=
(N)*S(N*(AN*S+L+LN*S+PN*S))*N*Ø
D/ASNØ/

D/ASNØ/=
\

CØNTAINS \
D/LN/

D/LN/=
((N)*S(N*(AN*S+L+LN*S+PN*S))*N*Ø+((N)*(AN*S+L+LN*S+PN*S)
(N*(AN*S+L+LN*S+PN*S))*N*Ø+(N)*Ø))
D/LA/

D/LA/=
(N)*S(N*(AN*S+L+LN*S+PN*S))*N*Ø
D/LS/

D/LS/=
(N*(AN*S+L+LN*S+PN*S))*N*Ø
D/LL/

D/LL/=
(\+(N)*S)(N*(AN*S+L+LN*S+PN*S))*N*Ø
D/LP/

D/LP/=
(N)*S(N*(AN*S+L+LN*S+PN*S))*N*Ø
```

Figure II.5 Continued

```
D/LØ/

D/LØ/=
\

CØNTAINS \
D/LNN/

D/LNN/=
((N)*S(N*(AN*S+L+LN*S+PN*S))*N*Ø+((N)*(AN*S+L+LN*S+PN*S)
(N*(AN*S+L+LN*S+PN*S))*N*Ø+(N)*Ø))
D/LNA/

D/LNA/=
(N)*S(N*(AN*S+L+LN*S+PN*S))*N*Ø
D/LNS/

D/LNS/=
(N*(AN*S+L+LN*S+PN*S))*N*Ø
D/LNL/

D/LNL/=
(\+(N)*S)(N*(AN*S+L+LN*S+PN*S))*N*Ø
D/LNP/

D/LNP/=
(N)*S(N*(AN*S+L+LN*S+PN*S))*N*Ø
D/LNØ/

D/LNØ/=
\

CØNTAINS \
```

Figure II.5 Continued

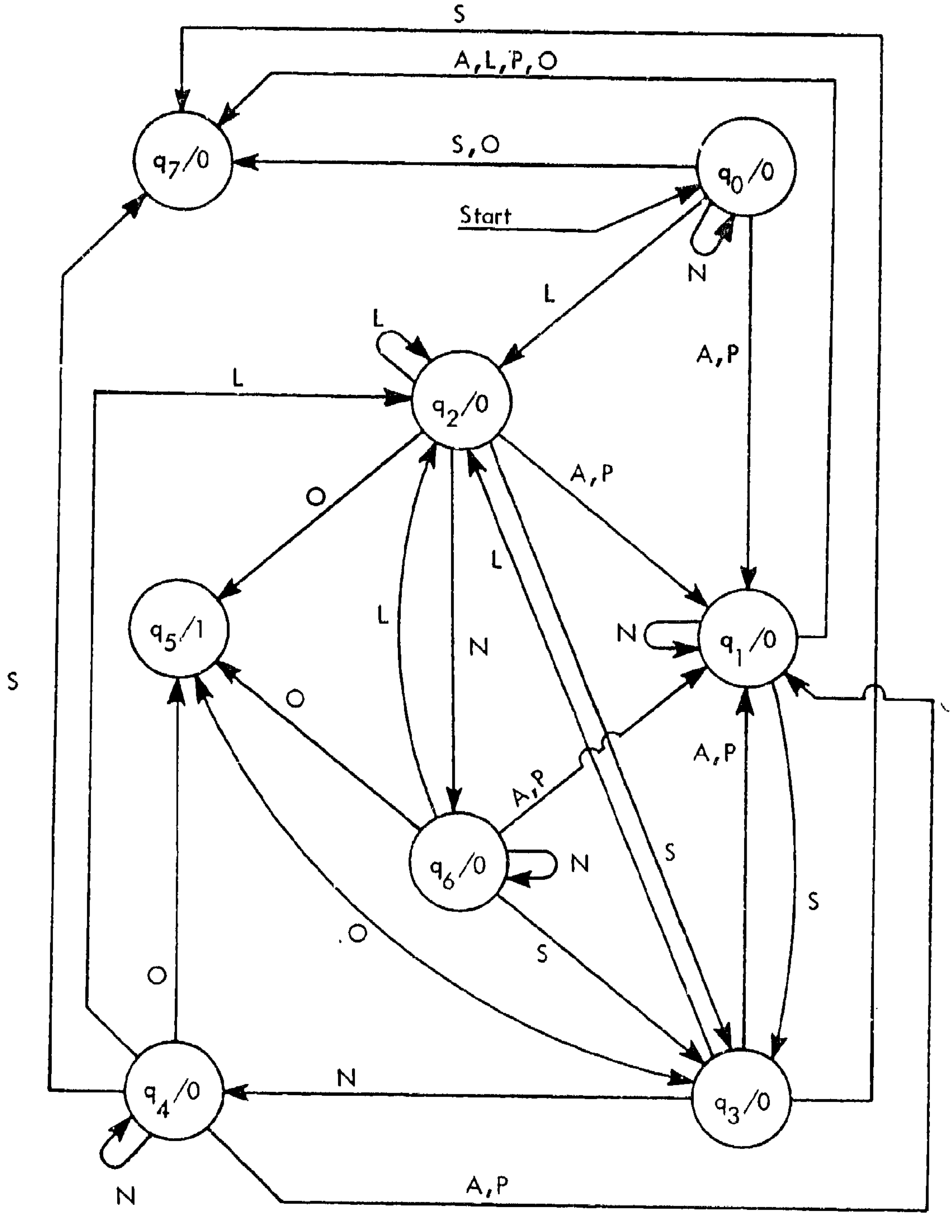


Figure II.6 Flow-graph for Machine to Find the Eta Function

will read the next character in the sequence. After this character is read it is tested by the subprogram to determine which subprogram the program will transfer to. The transfers from subprogram-to-subprogram are formed directly from the state-to-state transition as shown on the flow-graph. The terminal states (q_5 and q_7) are translated into subprograms which display the results of the test (e.g., a value is returned to a calling program or an appropriate message is printed).

CHAPTER III

STRING PROCESSING

The preceding chapters discussed the use of the regular expression and how the derivative can be formed by using manual manipulations of the regular expression. The problem is to write a program to perform these manipulations. The digital computer is designed to handle numerical formulas to which a numerical answer can be assigned. The problem $X = A + B(A + C)$, where values have been assigned to A, B, and C, can be easily solved on the computer by means of adding and multiplying the values of A, B, and C. The results of these operations are then placed in the location reserved for the answer, X.

In the case of the derivative of the regular expression we have a string of characters which represents the regular expression; in general this string is of unknown length. Before studying the process by which the computer forms the derivative the concept of string processing is introduced. String processing provides a method by which the computer can perform the basic manipulations on the string (e.g., storing, adding, or removing characters from a string) needed to form the derivative.

A string is defined as a series of characters that are written in a linear form (e.g., this sentence is a string). The symbol "$\not{S}$" will be used to represent the string. For this report only strings that are used to represent the regular expression will be considered. These are strings which contain only the following characters: the literals, $\emptyset$, λ, +, ., *, (,), [,], D, and R. These characters form the vocabulary of the string. Also the requirement of validity will be implied.

A valid string is one in which the operators are used properly. This implies that the parenthesis and brackets are properly formed and that the operators are used only in a valid context. A note should be made that the validity of a string is determined only by its context and not its content.

To aid in the discussion of the use of strings we introduce the terms: alpha character, operator, term, and level. The alpha character consists of the literals plus λ and ∅. The remaining characters (i.e., the +, ., *, (,), [,], and D) are the operators and delimiters.

A term is a logical unit of the string. It is a sub-string consisting of alpha characters and the star operator, or any sub-string which is contained within a set of parenthesis or brackets. If a term is followed by the star, the star is considered part of the term. The level provides a method of ranking the terms in a string. Terms that are connected by one of the operations of AND, OR, or concatenation are all of the same level. When several terms are grouped, via a set of parenthesis (or brackets), a new term is formed. The new term is of a higher level than the level of the individual terms. The phrase "to go down in level" means that, rather than testing the large term, the several terms that are enclosed in a set of parenthesis are to be tested.

The subroutine ∅SLVL has been written for REXPRO to perform the operation of reading a term. It returns with the first and last characters of the term. ∅SLVL also indicates if the term is followed by more terms of the same level.

A method is needed to store and operate on the strings in the computer. This is accomplished by a set of linked cells. A cell is simply a location which can hold one, and only one, character of the string.

To indicate how the individual cells are joined, to form the string, a linking system is used. The simplest linking system consists of linearly ordering the cells; i.e., the first cell is used to store the first character of the string, the second cell stores the second character of the string, etc.

In REXPRO the number of literals was restricted to 16 and two additional characters were added. These are the end flag (represented by the symbol "@") which is used to indicate the end of a string and the NULL character. Thus the vocabulary of REXPRO contains less than 32 separate characters. These characters can be encoded by a five bit binary word, Appendix A contains a list of the vocabulary and the codes used to store these characters.

The memory system of the SCC 650 consists of a main core with 4096, 12 bit words. For maximum efficiency, with respect to memory, two cells are stored in each memory word. To address a given cell a two-word addressing scheme is used. The MP (memory point) gives the address of the memory word and the HP (half point) indicates which cell in the work is being addressed. A HP equal to zero is used for the first cell and a HP equal to one is used to reference the second cell.

A prefix is used to denote the different locations in the string. These are LE, for level end; L, for level start; LR, for last read; R, for read; T and T2, for temporary; and W, for write.

To finish the description of string storage we must consider what happens when characters are added or removed from a string. This can be accomplished by shifting the characters in the string; however, this introduces the problem of having the end of the string blocked by another string.

The problem of removing characters is considered first. If part of the string is 'D ((WXYZ))' we see that the string contains an extra set of parentheses which should be removed. This string is shown in Figure III.1a as it would be stored in memory. Each horizontal block represents a memory word containing two cells. The number to the left gives the MP location of that word.

In the expansion[7] of the string the inner set of parentheses at '5001/0[8] (the MP/HP) and at '5003/1 are to be removed. When these are removed something must be placed in these cells so the string remains continuous. The new character is the NULL character. The NULL character has no value, but is simply a character used to fill out the string. Figure III.1b shows the string after the expansion with the symbol N being used to represent the NULL character.

The jump-flag is introduced for those cases where it is desired to add characters to the string. The jump-flag (or more simply the flag) is represented by placing a one in the sixth bit of the second cell in the memory word. In the examples it is indicated by using an F in the cell. Placing the flag in a cell indicates that the following word does not contain the remaining part of the string, but rather an address giving the location where the string is continued.

Figure III.2a shows the string 'D (ABC)@ ' as it appears in the memory. For this example it is desired to replace the character 'B' with the string 'WXYZ'. This is accomplished by removing the character 'B' and replacing it with 'F-W'

---

7 Any change in the string is called "expanding the string".

8 The apostrophe is used to represent number to the base eight, e.g., '5001 = $5001_8 = 2561_{10}$.

'5000	D	(
'5001	(	W
'5002	X	Y
'5003	Z	)
'5004	)	@

(a)

'5000	D	(
'5001	N	W
'5002	X	Y
'5003	Z	N
'5004	)	@

(b)

Figure III.1 The Use of the NULL Character.

(i.e., a flag and the symbol W) and saving the following two characters; the 'C)'. These characters are replaced with the address '5500 where the string is continued.

At location '5500/0 the remaining characters of the string are added along with the two characters, the 'C)', which are removed from the main string. At location '5502/1 a flag is placed in the string. The address in the following location refers back to the remaining portion of the string. The resulting storage in the memory is shown in Figure III.2b. This string is read as 'D(AWXYZC)@ '.

Several comments should be made about this example. First, as there is no space left in this method to store information about the half-point the requirement is added that all strings, and string segments, have to start with a half-point of zero. Second, a flag may be used only in the second cell; thus, we see the use of the 'F-N' (flag-NULL) at location '5502/1.

This process of removing a character in the string and adding an address is termed "setting a breakpoint". This is handled by the REXPRO subroutine $BPSET which not only adds the flag and jump address, but it also saves the characters that were removed from the string to give space for the address. The process of adding the jump at the end of the new string segment is termed "returning the break-point" and is executed by the subroutine $RTBP. This routine also adds the characters which were removed from the string by $BPSET.

The subroutines $READ (string read) and $WRITE (string write) are two additional routines which are used in processing the string. These four routines form the bases for all of the string operations performed by REXPRO. A description of the subroutines $BPSET, $RTBP, and $SLVL is given in Appendix D.

'5000	D	(
'5001	A	B
'5002	C	)
'5003	@	

(a)

'5000	D	(
'5001	A	F-W
'5002	'5500	
'5003	@	
'5500	X	Y
'5501	Z	C
'5502	)	F-N
'5503	'5003	

(b)

Figure III.2 An Example of the Use of the Jump-Flag.

CHAPTER IV

USING THE COMPUTER TO FORM THE DERIVATIVES

The previous chapters introduced the derivative of the regular expression and the concept of string processing. We are now ready to study the processes by which the computer forms the derivative. The computer operation will be studied by following an example through the steps needed to form the derivative.

The strategy used in forming the derivative is to repeatedly apply the following rules until all of the terms are of the form given in Rule IV.7b. These rules are,

IV.1 $D((\$)^*) = D((\$))(\$)^*$

IV.2a $D(\$_1 \$_2) = D(\$_1)\$_2$, if $\eta(\$_1) = \emptyset$

IV.2b $D(\$_1 \$_2) = (D(\$_1)\$_2 + D(\$_2))$, if $\eta(\$_1) = \lambda$

IV.3 $D(\$_1 + \$_2) = (D(\$_1) + D(\$_2))$

IV.4 $D([\$]) = [D(\$)]$

IV.5 *

IV.6 $D((\$)) = D(\$)$

IV.7a $D(A_1{}^*A_2B_1 \text{---} B_n) = D((A_1)^*A_2B_1 \text{---} B_n)$,

where A_1, A_2 are literals and $B_1, \ldots B_n$ are literals or the star operator.

IV.7b $D(A_1A_2B_1 \text{---} B_n)$, where A and B are as defined above.

IV.8 $D(\$_1 \cdot \$_2) = (D(\$_1) \cdot D(\$_2))$

* The first eight rules are numbered according to the subroutines which use these rules. Due to a reorganization of the subroutines there is no rule IV.5.

After all of the terms are of the type given in Rule IV.7b the following rules are used as the final step in forming the derivative.

IV.9a $\quad D/A/\,(A_1A_2\text{---}A_n) = \lambda(A_2\text{---}A_n)$, if $A = A_1$

IV.9b $\quad D/A/\,(A_1A_2\text{---}A_n) = \emptyset$, if $A \neq A_1$

We should note that these rules differ slightly from those given in Chapter II, but they are derived from the rules in Chapter II. These rules were modified so that they would be compatible with the string processing used in REXPRO.

Expanding The Derivative

We will now consider the example D/A/ (R), where R = (A*B + C)*AB*. This would be stored in the computer as,

'D ((A*B + C)*AB*)@ '

and the "name of the derivative" is 'A'. It should be noted that the derivative name is not stored as part of the derivative; e.g., as a subscript. Rules IV.1 through IV.8 are not a function of the particular derivative being formed; thus, the name can be separated from the derivative and retrieved only where it is needed.

It is also noted that the derivative is stored with 'D (' and ') ' added to the string. This provides easily detected delimiters to outline the derivative. A search is made of the term(s) following the 'D (' and a list of the characteristics of these terms is made. For this example we find the two terms '(A*B + C)*' and 'AB*'. Another check is made and we find that the terms are connected by

juxtaposition, i.e., the terms are concatenated and Rule IV.2 applies. To determine if Rule IV.2a or IV.2b is to be used the eta function must be evaluated. The method of evaluating eta will be deferred until the end of the chapter, but it is found that η ((A*B + C)*) = λ. Thus the expression,

'(D ((A*B + C)*)AB* + D(AB*))@ '

is formed by applying Rule IV.2b.

It is pointed out that the expansion of the string, produced by Rule IV.2b, is formed by using the techniques presented in Chapter III; i.e., adding and removing characters from the string and copying portions of the string. The exact methods that are used in the expansion process are reserved for the description of the subroutine ØDF2 in Appendix D.

The expression is still not of the form given in Rule IV.7b, thus the process is repeated. The string is scanned from left to right until the first delimiter, the 'D(', is found. The following operand, '(A*B + C)*, is then read. This operand consists of a single term and that term is operated on by the star. Thus Rule IV.1 is used to produce the expanded string,

'(D ((A*B + C)) (A*B + C)*AB* + D (AB*))@ '

The scanning continues until the second delimiter is found. The following term, 'AB*, is then read. This operand appears to follow Rule IV.1 as it contains only one term and is followed by a star. The question arises as to what characteristic will differentiate between terms of the form '(A*B + C)*' and the

form 'AB*'. The difference lies in the fact that in the first case the entire term is operated on by the star, while in the second case only the one literal is operated on by the star. This characteristic is easily checked by determining if the first character in the term is '('. The routine which reads the term (the subroutine ØSLVL) saves the first character for this test. We see that 'AB*' does not follow Rule IV.1, but Rule IV.7. The subroutine ØDF1, which is used to expand this form, finds that the first literal is not operated on by a star. Under this condition Rule IV.7b is invoked and no change is made.

The scan is continued and no more derivative delimiters are found before the end flag (the '@ '). Thus a new pass is started and the process is repeated. The first operand which is found is '(A*B + C)'. This has the characteristics of having only one term, not operated on by a star, and it is enclosed in parentheses. These characteristics match those of Rule IV.6, thus the result,

'(D (A*B + C) (A*B + C)*AB* + D (AB*))@ '

The scan is continued and the only term found is 'AB*' to which Rule IV.7b is applied and no change is made.

A new pass is then started and the operand 'AB* + C' is found. This operand consists of the terms 'A*B' and 'C' and they are connected by the OR operator. For this condition Rule IV.3 is used to produce,

'((D (A*B) + D (C)) (A*B + C)*AB* + D (AB*))@ '

Again no change is made in the last operand and a new pass is started. The first operand found is 'A*B'. This operand consists of a single term with the first character a literal. Thus Rule IV.7 is used. The subroutine which processes this rule scans the term and finds that the first character is operated on by the star and Rule IV.7a is invoked. This produces the string,

'((D ((A)*B) + D (C)) (A*B + C)*AB* + D (AB*))@ '

The process is repeated through several more passes and the following expression is obtained.

'(((D (A) (A)*B + D (B)) + D (C)) (A*B + C)*AB* + D (AB*))@ '

A final pass is then made. During this pass all of the operands are found to follow Rule IV.7b. Thus no changes are made in the string. This indicates the end of the expansion operations. The final step in obtaining the derivative is to implement Rules IV.9a and IV.9b. These rules are implemented by the subroutine ØMATCL. In this routine a single scan is made of the derivative. When the derivative delimiter is found the following literal is compared with the name of the derivative (in this example the literal A). If the two characters match, the string is modified by Rule IV.9a, otherwise the Rule IV.9b is used. This produces,

'(((λ (A)*B + Ø) + Ø) (A*B + C)*AB* + Ø) @ '

After this derivative is formed the following rules are used to simplify the results.

IV.10 $\quad \emptyset\$ = \$\emptyset = \emptyset$

IV.11 $\quad \emptyset + \$ = \$ + \emptyset = \$$

IV.12 $\quad \lambda\$ = \$\lambda = \lambda$

IV.13 $\quad \emptyset . \$ = \$. \emptyset = \emptyset$

The simplifications are performed in a manner to similar to the operation of expanding the string; i.e., repeated passes are made until a pass is found in which no changes were made. The final result is,

D/A/ (R) = '(A)*B (A*B + C)*AB* @ '

To summarize the operation of forming the derivative:

A. Search the string to find the derivative delimiter and test the following operand. From this test determine which rule applies. Appendix D describes in detail the subroutine $\$$DCLAS which performs this operation.

B. Expand the string according to the appropriate rule. The subroutines $\$$DF1, $\$$DF2, $\$$DF38, $\$$DF4, $\$$DF6, and $\$$DF7 are used to implement the rules.

C. Repeat A and B until a pass is made with no change in the string.

D. The derivative is taken by the subroutine $\$$MATCL which implements Rules IV.9a and IV.9b.

E. Simplify the string.

In this example the derivative was taken with respect to the single character 'A'. If the derivative with respect to a string of characters is desired, for example D/AB/ (R), the rule D/AB/ (R) = D/B/ (D/A/ (R)) is used. In other words the

derivative, D/A/ (R), is formed as given above. Then the process is repeated, using the above results, to form the derivative with respect to B.

Calculating the Eta Function

A technique is now introduced which will allow the eta function to be calculated by the computer. The eta function is defined as,

IV.14 $\eta(\not{S}) = \begin{cases} \lambda, & \text{if } \lambda \in \not{S} \\ \emptyset, & \text{if } \lambda \notin \not{S} \end{cases}$

IV.15 $\eta(\not{S}_1\not{S}_2) = \eta(\not{S}_1) \cdot \eta(\not{S}_2)$

IV.16 $\eta(\not{S}_1 + \not{S}_2) = \eta(\not{S}_1) + \eta(\not{S}_2)$

IV.17 $\eta(\not{S}_1 \cdot \not{S}_2) = \eta(\not{S}_1) \cdot \eta(\not{S}_2)$

IV.18 $\eta([\not{S}]) = [\eta(\not{S})]$

From the first rule we see that the eta function is a binary function; i.e., it has only two values λ and $\emptyset$. The other rules state that any eta function can be formed by finding the eta function for simplier terms.

We first consider the types of expression for which the eta function can be found directly.

A. A simple term consisting only of alpha characters and the star operator; e.g., A*B, ABC, or A*B*. For this case we consider each alpha character at a time.

$\eta(A^*) = \lambda$, by the definition of the star operator

$\eta(A) = \emptyset$, A any literal or $\emptyset$

$\eta(\lambda) = \lambda$, by definition of the eta function

For several characters which are concatenated we use Rule IV.15

and the fact that $\lambda . \emptyset = \emptyset$. From this we conclude that the eta value of a simple term equals lambda if, and only if, each alpha character is a lambda or is operated on by a star. This test can be easily performed on the computer by scanning each character from left to right and testing the characters to see if they are lambda or if they are alpha characters followed by a star. This is the method that was presented as the second example problem in Chapter II.

B. A complex term of the form $(\mathcal{S})^*$. Again by the definition of the star operator we find that $\eta((\mathcal{S})^*) = \lambda$.

C. A complex term of the form $(\mathcal{S})$. For this case the eta function cannot be directly determined; i.e., the terms enclosed in the parenthesis have to be tested to find the resulting eta function.

This process of testing the terms was written into the subroutine ØBETAT. This subroutine returns the three results, the eta value equals lambda, equals phi, or is undetermined.

For the cases with more complex expression we will use the following rules.

IV.19 $\lambda . X = X$

IV.20 $\lambda + X = \lambda$

IV.21 $\emptyset . X \doteq \emptyset$

IV.21 $\emptyset + X = X$, where X is either lambda or phi

These rules are derived from the fact that the values of the eta function (i.e., lambda or phi) are binary values.

With these rules in mind we consider the example $\eta((A^* + B) A + C^*)$. This would be stored in the computer as '($(A^* + B) A + C^*$)' with a pointer to the first left parenthesis at the start. The term '($(A^* + B)A + C^*$)' is tested and it is found to be undetermined. Thus it is necessary to "go down in level" to evaluate the expression. This is performed by stepping the pointer one location to the right. The term '$(A^* + B)$' is then tested and it is also found to be undetermined.

The process of going down in level is repeated and the term 'A*' is tested and found to have an eta value of λ. This term is followed by the OR operator. From equation IV.20 we see that result will be lambda, even without testing the remaining term. Thus the term 'B' is passed over without being evaluated and the operator ')' is read. This indicates the end of the present level and the need to determine "what to do next".

In this example the term '(A* + B)' was found to have a value of lambda and it is concatenated with the following term (the 'A'). By equation IV.15 we see that in evaluating concatenated terms they are treated as being "AND'ed". Thus by IV.19 we find that the value of expression '(A* + B) A' is determined solely by the term 'A'. The term 'A' is tested and found to have a value of phi.

The value of that part of the expression which has been evaluated is phi and the following operator is OR and by IV.22 we see that the result is determined solely by the following term. This term, the 'C*', is tested and found to have a value of lambda. The only remaining character is a right parenthesis which indicates the end of the expression being evaluated. Therefore, the eta value of the expression being tested is λ.

This example has shown a method by which the eta function can be found by a process of scanning the expression from left to right and using Rules IV.19 to IV.22 to determine which terms are tested and which are passed over. While this example was admittedly simple the same technique can be used to evaluate any expression. For the interested reader Appendix D describes the operation of the

subroutines ØETATT and ØBETAT. The subroutine ØETATT performs the operation of deciding "what to do next". ØBETAT is called by ØETATT to evaluate single terms.

CHAPTER V

REXPRO OPERATION

Memory Organization

The 4096, 12 bit, word memory of the SCC 650 is divided into three distinct areas for this program, as shown in Figure V.1. The first section consists of the first 64 words. These words can be addressed from the entire computer memory via the direct memory instructions.

The direct memory area is further divided into four sub-sections. The first section occupies locations '0 to '17 and holds the symbol table. This table stores the literals that have been entered. The literals are stored as trimmed ASCII characters. These literals are stored in the string as a four bit character which corresponds to their location in the table. A flag is placed in the seventh bit if the character was seen as a part of the regular expression string as opposed to a substitution string.

The second section of the direct memory contains the substitution table and it is stored from '20 to '37. This table contains the addresses of the characters which are placed in the regular expression string as the name of substitution strings which are to be substituted into the main string. The table is referenced by adding '20 to the location in the symbol table where the corresponding character is stored. For example, if the character 'A' is the third character which is stored in the symbol table it will be replaced with '02 and its substitution string address will be stored at '22. If no substitution string has been stored the corresponding location in the substitution table contains '0000.

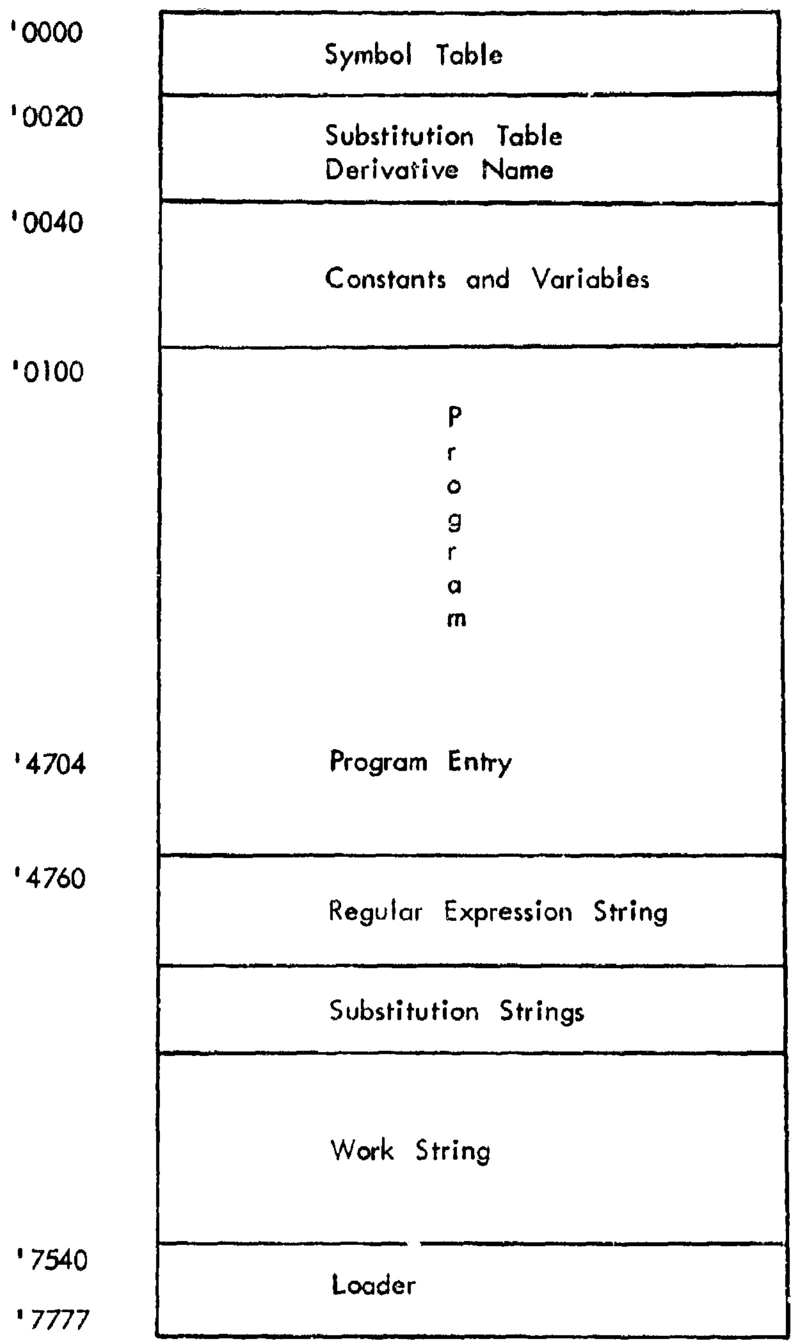


Figure V.1 Memory Allocations

The third section contains the converted (via the symbol table) literals that are used to name the derivatives. As this data is used at a point in the program execution at which the substitution table has already been used the derivative name is also stored from '20 to '37. The derivative name is stored as a string, packed two characters per word. Therefore, the derivative name is also referred to as the derivative string. A maximum length of 31 characters can be used to specify the derivative name (one cell being reserved for an end flag).

The last section of the direct memory (from '40 to '77) contains the non-string variables. The information contained in this section consists of string starting addresses, string working addresses, temporary character and address storage, input mode information, and a program start address. A detailed list of the information stored in this section is contained in Appendix B.

The next section of the memory is allocated to the storage of the program itself. This section runs from '100 to '4757. The program starts execution at '4704.

The remainder of the core is used to store strings. The first string which is stored is the regular expression string. This is followed by the substitution strings, if any. The last string is the work string. The work string is a copy of the regular expression string, with substitutions made. The work string is the string which is expanded to form the actual derivative.

There is no fixed length for these strings. The only limitation on length is the fact that they have to fit the area from '4760 to '7540. (The upper limit was designed to save the loader. It can be increased to give more working area or decreased to save other programs).

Program Execution

The program is executed in four distinct phases as shown in Figure V.2.

Phase I. During the first phase (controlled by MAINLINE) the program initializes the direct memory by clearing the symbol table, the substitution table, and the string addresses. This phase also sets the input mode for the teletype keyboard, loads the program starting address (in the variable RESTRT) and sets NEW. NEW defines the free area where strings can be stored. At the end of Phase I the message "R-EXP PROGRAM--READY" is printed and the program enters Phase II.

Phase II. This phase is the main input phase and is controlled by the routine MAINLINE. During this phase either control words or strings may be entered into the computer. The program receives the first character from the teletype which is not a carriage return, line feed, space, or tape leader. A second character is then received. If the second character was a letter it is assumed to be part of a control word. Otherwise it is treated as a string. The first two characters of the control word are stored. When a line feed is received the two characters are tested and the control word is executed. The functions of the control words are listed below:

"TAPE" - Input via paper tape
"KEYBOARD" - Input via the keyboard
"EXECUTE" - Enter Phase III
"TERMINATE" - Return to Phase I

If the operator makes a mistake or desires to make a change he may type a left arrow before the line feed. The operator can then enter a new string or control word. The entering of an incorrect control word will cause the following to be printed: "UNDEFINED CONTROL WORD".

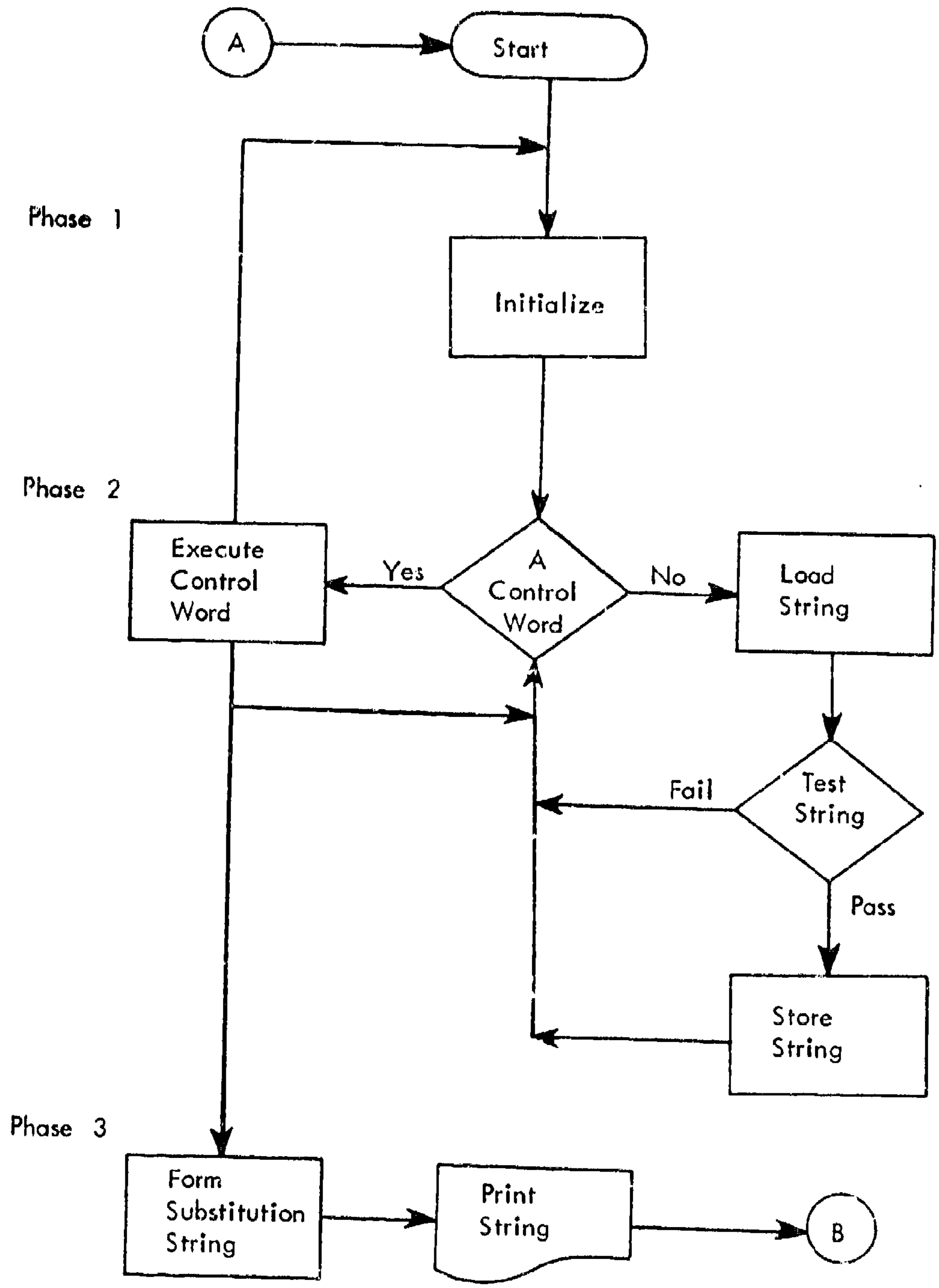


Figure V.2 Flow-graph of Program Execution

Phase 4

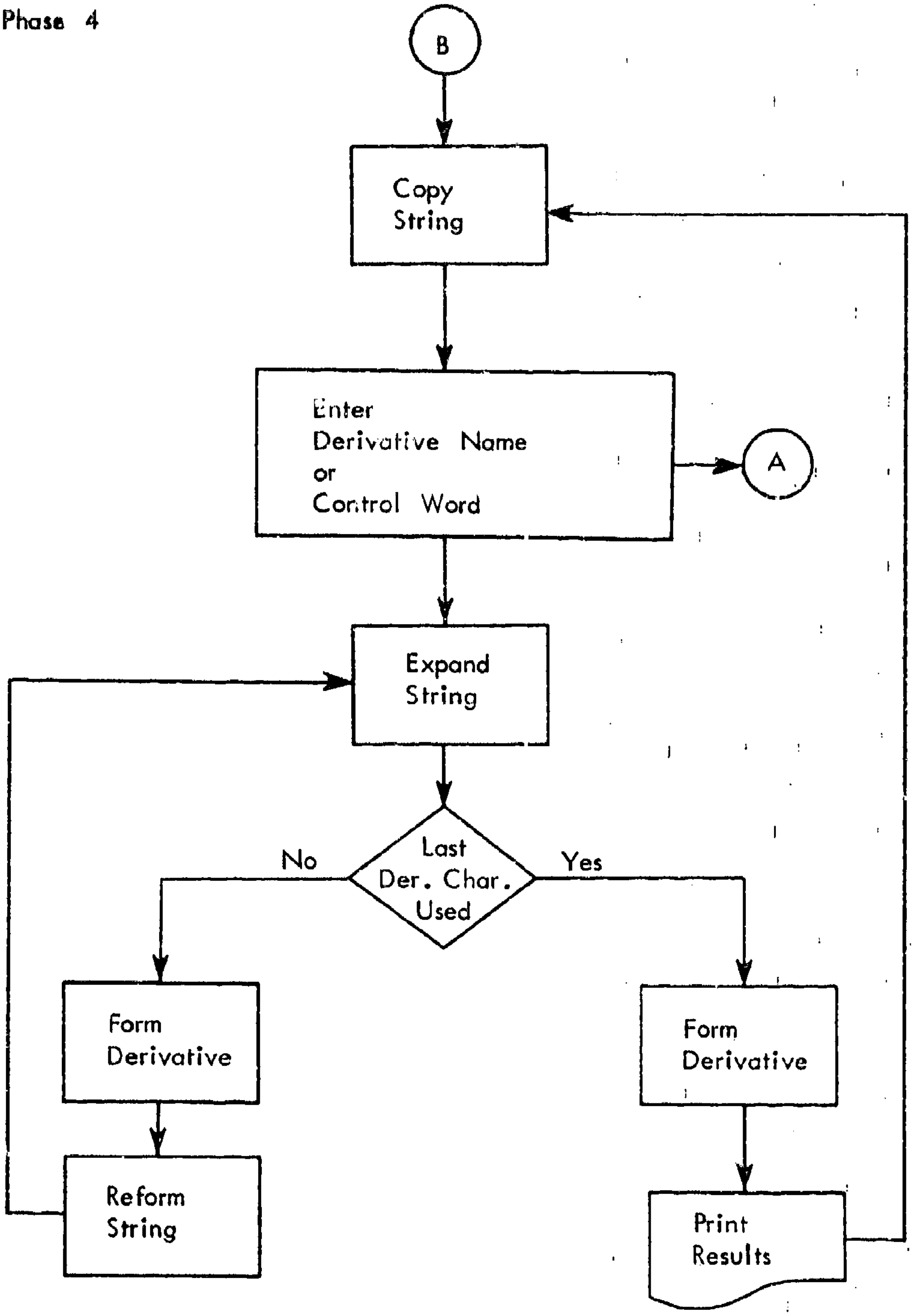


Figure V.2 Continued

If the second character entered was not a letter, the program checks to see if it is the start of the string. The first string that is stored must be the regular expression string and it is denoted by an 'R', followed by optional spaces, followed by an '='. If the string does not contain an '=' the message "STRING DOESN'T CONTAIN = N.S." is printed. The "N.S." indicates that the string is not stored.

After the equal mark any valid input character may be entered. These characters are listed in Appendix A along with the code in which they are stored. An invalid character will cause a '?' to be printed and the character is not stored. To aid in the loading of a string, a carriage return, a line feed, or a space may be entered at any time after the equal sign.

To correct mistakes in loading a string the operator may use a left arrow or a vertical arrow. The left arrow dumps the string that was being stored and allows a new string, or control word, to be entered. The vertical arrow removes the last character(s) which were entered; one character is removed for each vertical arrow. The vertical cannot remove the characters to the left of the string; i.e., the term 'R='. These editing aides must be used before the string is terminated.

The actual storage of the string is handled by $STORE and it is stored at the location given by NEW. The program adds a 'D (' prefix to the string and a ')' suffix. These characters are needed for proper execution of the string.

An '@' is entered to terminate the string. At this point the $OPCK, $PBCK, and $AOCK subroutines are called to test the strings. They test for the proper context of the various operators, for properly formed parentheses and

brackets, and for use of the ill-defined and-or operation (e.g., A + B.C). If any of these errors are found, the respective messages "ILL-FORMED OPERA. -N.S.", "ILL-FORMED ( ) OR [ ] -N.S.", are printed. It should be noted that the testing is halted when the first error is found. If no errors are found the string is "stored" by updating NEW and storing the starting location of the string. Also all of the literals that were entered with the regular expression strings are given a flag in the seventh bit.

If there was an error, a new string may be entered; this string is written over the old string. A new string may be entered to replace a string which has already been stored. This does not remove the original string, but sets a pointer to the new string.

After the regular expression string has been stored substitution strings may be entered. These are entered in the form 'A = -----'; where the 'A' is the literal in the regular expression string which is to be replaced by the string to the right of the equal mark. Only one level of substitution is permitted; thus the name of the substitution string (e.g., 'A') must have been entered in the symbol table and it must also have a flag set in the seventh bit. If this is not met the message "SUBSTITUTION NOT SEEN-N.S." is printed. Except for this test and the fact that the string is prefaced by '(' instead of 'D(' the loading, testing, and storing of the string is the same as the regular expression string.

When the strings are formed the subroutine which forms the symbol table checks to see if it is full (more than 16 entries). If it is full the error message "TABLE FULL--EXEC. HLT." is printed. Likewise during the loading of the

string (or during any other phase in which characters are being written in the string) the program checks to see if there is room left in the memory. If not "MEMORY FULL-EXEC. HLT." is printed. (The upper limit of the available memory, a normal '7540, is stored at location '0764 and can be changed. It should be set at five less than the desired upper limit). Both of these errors are unrecoverable so the program automatically re-enters Phase I.

Phase III. When the program enters Phase III the routine ØSUB is called. This routine inserts the substitution string, if any, in the regular expression string. A copy of the regular expression string is then printed so that the operator can check the string. If no string was stored a '?' will be printed and control returns to Phase II, otherwise the program goes to Phase IV.

Phase IV. This phase performs the actual operation of taking the derivative and it is controlled by DCONT. This phase starts by calling ØDRCPY to form a work copy of the regular expression string.

The subroutine ØDIN is called. This is a specialized form of MAININ. The only control word that can be entered is "TERMINATE" and it is tested when a line feed is received. This control word causes the program to return to Phase I.

If the first character that was entered was a 'D' then the program enters a derivative name storage section of ØDIN. The name has to be prefixed by 'D/'. and then the operator inputs the string of characters which forms the derivative name. The only valid characters are the literals which were entered in the regular expression string. An invalid character will cause a '?' to be printed. A test of the length of the derivative name is made and if it exceeds 31 characters

then "DERIVATIVE Ø FULL-N.S." is printed and the name is not stored. During this phase all of the information is received via the keyboard.

As in Phase II a left arrow and a vertical arrow can be used to edit the name or the control word. When the name is followed by a '/' the routine returns to DCONT. If no string has been stored a '?' will be printed and the routine remains in ØDIN.

When the program returns to DCONT the subroutine ØDCLAS is called to classify the forms of the derivative of the string. ØDCLAS inturn calls ØDF1, ØDF2, ØDF38, ØDF4, ØDF6, and ØDF7 to expand the string.

After one pass through the work string the program returns to DCONT. At this point a check is made to see if any changes were made in the work string. If a change was made, then the program returns to DCONT.

This is repeated until a pass is made with no changes in the string. At this point the string consists only of terms of the type 'D(' followed only by literals which are in turn followed by a ')'. These terms can be combined by any of the valid regular expression operators.

DCONT then finds the first character in the derivative name and passes this to ØMATCL. This routine has two purposes. The first is to perform the final operation of taking the derivative. This uses the rules:

$$D_A (A. . . .) = \lambda$$

$$D_B (A. . . .) = \emptyset$$

ØMATCL then calls four subroutines to simplify the string using the regular

expression identities. As in $DCLAS this is an iterative operation which is repeated until there are no changes in the string.

DCONT then reforms the work string by adding 'D(' to the front and ')' to the end as these were removed by $MATCL. The program then goes back to $DCLAS and the procedure is repeated using the second character in the string name. The total process is repeated until the last character in the derivative name is used. When this occurs the results are printed along with the name of the derivative. $ETATT is then called to see if the string contains lambda and it prints "CONTAINS \" if the test passes.

At this point one complete derivative has been found and DCONT returns to the routine $DRCPY to obtain a new work string and the process is repeated. The program remains in Phase IV until the control word "TERMINATE" is entered.

(The user who wants to study the operation of the program can replace the following "NOPS" with calls to the printing routines. At location '4554 load '5301 and at '4555 load '1200. This causes the string to be printed for every pass through "$DCLAS". At location '4610 he can load '5301, at location '4611 load '1233, at '4612 load '5301, and at '4613 load '1200. This causes the derivative name and the string to be printed for each pass through MATCL. Due to the large amount of data that is produced and the slow speed of the printer these changes are not recommended for normal use).

CHAPTER VI

CONCLUSION

This paper has described a practical digital computer program which can be used to find the derivative of a regular expression. Chapter II shows how useful the regular expression can be in the design of a finite state, sequential machine. The remaining chapters describe the techniques used to implement REXPRO and its use.

The usefulness of this program can be seen by studying the second example in Chapter II. The optional prints were added, as described in Chapter V, to print the partial results for every pass through the subroutine ØDCLAS. These partial results are formed in approximately the same manner in which a designer would form the derivative. The partial results, shown in Figure VI.1, were those obtained during the process of calculating the derivative D/A/R. When one considers the amount of work needed to form this derivative, and that this is only one of the several derivatives that are needed for this example, this program drastically reduces the amount of work (and thus the possibility of errors) needed to use the regular expression in design work.

This example shows one of the problems that are involved in using the derivative of the regular expression. In the flow-graph which describes the machine designed for example two, Figure II.6, it can be shown that state q_2 and q_6 are identical. This implies that the derivatives associated with these states are also identical.

These derivatives are,

D/L/ = (/ + N*S) (N* (AN*S + L + LN*S + PN*S) (*N*O, and

D/LN/ = (N*S (N* (AN*S + L + LN*S + PN*S)) *N*O + (N* (AN*S +L + LN*S
+ PN*S) (N* (AN*S + L + LN*S + PN*S)) *N *O + N*O))

which are not obviously the same.

The question appears at this point; how can these two derivatives be tested to show that they are the same? Can this testing be done by the program so that the operator does not have to monitor the operation of the program? Can we guarantee that a machine will be formed with a minimum number of states?

Another question arises from this work. Can the regular expression be forced to generate a machine of a specific form? This machine would not necessarily be a minimal state machine, but rather a machine with certain desired characteristics; e. g., a machine that is easily implemented by a particular type of hardware.

It is hoped that by using this program as a tool that these questions can be answered.

```
R-EXP PRØGRAM--READY
R=N*Z(N*Z)*N*Ø@
Z=AN*S+L+LN*S+PN*S@
EXECUTE

CHECK CØPY..
D(N*(AN*S+L+LN*S+PN*S)(N*(AN*S+L+LN*S+PN*S))*N*Ø)

D/A/

(D(N*)(AN*S+L+LN*S+PN*S)(N*(AN*S+L+LN*S+PN*S))*N*Ø+D((AN
*S+L+N*S+PN*S))(N*(AN*S+L+LN*S+PN*S))*N*Ø)

(D((N)*)(AN*S+L+LN*S+PN*S)(N*(AN*S+L+LN*S+PN*S))*N*Ø+D(A
N*S+L+LN*S+PN*S)(N*(AN*S+L+LN*S+PN*S))*N*Ø)

(D((N))(N)*(AN*S+L+LN*S+PN*S)(N*(AN*S+L+LN*S+PN*S))*N*Ø+
(D(AN*S)+(D(L)+(D(LN*S)+D(P(N)*S)))))(N*(AN*S+L+LN*S+PN*S
))*N*Ø)

(D(N)(N)*(AN*S+L+LN*S+PN*S)(N*(AN*S+L+LN*S+PN*S))*N*Ø+(D
(A(N)*S)+(D(L)+(D(L(N)*S)+D(P)(N)*S))))(N*(AN*S+L+LN*S+PN
*S))*N*Ø)

(D(N)(N)*(AN*S+L+LN*S+PN*S)(N*(AN*S+L+LN*S+PN*S))*N*Ø+(D
(A)(N)*S+(D(L)+(D(L)(N)*S+D(P)(N)*S)))(N*(AN*S+L+LN*S+PN
*S))*N*Ø)

(D(N)(N)*(AN*S+L+LN*S+PN*S)(N*(N*S+L+LN*S+PN*S))*N*Ø+(D
(A)(N)*S+(D(L)+(D(L)(N)*S+D(P)(N)*S)))(N*(AN*S+L+LN*S+PN
*S))*N*Ø)

D/A/=
(N)*S(N*(AN*S+L+LN*S+PN*S))*N*Ø
```

Figure VI.1 The Derivative D/A/R

BIBLIOGRAPHY

BIBLIOGRAPHY

REFERENCES SIGHTED IN TEXT

1. Booth, T. L., *Sequential Machines and Automata Theory*, John Wiley and Sons, New York, Chapter VI, pp. 214-243, 1967.

2. Brzozowski, J. A., "Derivatives of Regular Expressions", *J. Assoc. Comp. Mach.*, Vol. 11, pp. 481-494, October 1964.

3. Ghiron, H., "Rules to Manipulate Regular Expressions of Finite Automata", *IRE Trans. Electronic Computers*, Vol. EC-11, pp. 574-575, August 1962.

ADDITIONAL REFERENCES

4. Arden, D. N., "Delayed Logic and Finite State Machines", *Proc. AIEE Second Annual Symposium on Switching Circuit Theory and Logical Design*, Detroit, Michigan, pp. 133-151, October 1961.

5. Brzozowski, J. A., "A Survey of Regular Expressions and State Graphs for Automata", *IRE Trans. Electronic Computers*, Vol. EC-11, pp. 324-335, June 1962.

6. Brzozowski, J. A., and McCluskey, E. J., "Signal Flow Graph Techniques for Sequential Circuit State Diagrams", *IEEE Trans. Electronic Computers*, Vol. EC-12, pp. 67-76, April 1963.

7. Elgot, C. C., and Rutledge, J. D., "Operations on Finite Automata", *Proc. AIEE Second Annual Symposium on Switching Circuit Theory and Logical Design*, Detroit, Michigan, October 1961.

8. Kleene, S. C., "Representation of Events in Nerve Nets and Finite Automata", *Automata Studies, Ann. Math. Studies*, Princeton, New Jersey, p. 129, 1956.

9. McNaughton R., and Yamda, H., "Regular Expressions and State Graphs for Automata", *IRE Trans. on Electronic Computers*, Vol. EC-9, pp. 39-47, March 1960.

APPENDIX

APPENDIX A

PROGRAM ALPHABET AND CODING

The following list defines all of the characters that may be used to enter regular expressions in the computer, their meaning, and coding. The ASCII code is the value that is received from the teletype or used to output on the teletype. The internal code is the value used to represent the characters in the program.

Character	ASC II	Internal Code	
A - C E - Q S - Z O - 1	'01-'03 '05-'21 '23-'32 '60-'61	'0 - '17	Literals Assigned by Symbol table
(	'50	'20	
)	'51	'21	
[	'33	'33	"NOT" delimiter
]	'35	'35	
\	'34	'34	Lambda
-	'55	'25	Ø
D	'04	'30	Derivative
R	'22	not stored	R-EXP
=	'75	not stored	
.	'56	'26	"AND"
@	'00	'37	End flag
/	'57	'27	Derivative delimiter
*	'52	'22	Star operator
NULL	internal	'36	NULL character
+	'53	'23	"OR"

APPENDIX B

LIST OF VARIABLES

The following list gives all of the variables used in REXPRO, along with their location and a description of their meaning. All of these variables are stored in the direct reference portion of the SCC 650 memory.

APLUS	'45	'A' temporary storage
ATEMP	'45	'A' temporary storage
BC	'44	Brackets counter
CHRMAT	'44	Character match
DCTM	'44	Derivative counter
DRSTRT	'75	Start of derivative $\not{S}$
DPC	'44	Derivative pass counter
ETAVAL	'45	ETA value, η ($\not{S}$)
LC	'43	Line counter
LEHP	'66	Level end half point
LEMP	'67	Level end memory point
LHP	'64	Level start half point
LMP	'65	Level start memory point
LRHP	'62	Last read half point
LRMP	'63	Last read memory point
MESST	'53	Message address
NEW	'73	Next available $\not{S}$ storage location
PC	'43	Parenthesis counter
RESTRT	'77	Start of program
RHP	'60	Read half point
RMP	'61	Read memory point

RSTART	'74	Start of R-EXP Ø
SYMS	'56	Location in symbol table
SA1	'50	Ø add one at end of break point
SA2	'51	Ø add two at end of break point
SRP	'52	Ø return address at end of break point
TIN	'72	Type of input
THP	'54	Temporary half point
TMP	'55	Temporary memory point
T2HP	'56	Sec. temporary half point
T2MP	'57	Sec. temporary memory point
WHP	'70	Write half point
WMP	'71	Write memory point
XPLUS	'46	'X' temporary storage
XTEMP	'47	'X' temproary storage

APPENDIX C

PROGRAM LOADING ORDER AND MAP

This list gives the order in which the various subroutines, and the mainline program, for REXPRO are loaded in the memory. For those interested in studying the operation of the subroutines the starting address of each of the subroutines is given. These addresses are given as octal (base eight) numbers.

ANORTT	0100
BIN	0117
BOUT	0140
CONVRT	0153
DERTST	0211
ENDTST	0223
LITTST	0235
LLTST	0246
NULTST	0265
RLTST	0277
STRTST	0316
SYMF	0330
CRLF	0353
MESS	0401
$TRD	0666
$RDNNL	0676
$READ	0705
$NWRT	0737
$REWRT	0747
$WRITE	0760

ALPHIT	1022
READ1	1044
QUES	1076
SYMIN	1112
SYMSRH	1127
CONSYM	1153
%OUT	1200
DTITLE	1233
%SLVL	1306
%1SIM	1416
%2SIM	1525
%3SIM	1636
%4SIM	1724
%OPCK	1765
%COPY	2040
%AOCK	2061
%MATCL	2175
%DIN	2317
%PBCK	2506
%BPSET	2616
%RTBP	2724

ØDRCPY	2761
ØSUB	3003
ØSTORE	3045
ØBETAT	3241
ØETATT	3335
ØDF2	3500
ØDF7	3655
ØDF6	3725
ØDF4	3751
ØDF38	4033
ØDF1	4143
ØDCLAS	4213
MAININ	4346
DCONT	4542
MAINLINE	4704

APPENDIX D

DESCRIPTION OF SELECTED SUBROUTINES

In this appendix a detailed description is given for the subroutines: ØSLVL, ØBPSET, ØRTBP, ØDCLAS, ØDF1, ØDF2, ØDF38, ØDF4, ØDF6, ØDF7, ØETATT, and ØBETAT. The first three subroutines are used to perform basic string manipulations. The remainder are used to form the derivative of the regular expression. These subroutines were selected to describe the basic operation of REXPRO without getting involved in the numerous, but necessary, subroutines that perform secondary functions; e.g., the routines used for input/output.

These subroutines are described by means of a flowlist. The flowlist presents the various steps involved in executing that routine. The symbols "A" and "X" are used to represent the accumulator and index registers. The representations, (LRMP)*, indicates an indirect operation; i.e., the value stored at the location LRMP is used as a pointer to the desired location. The remaining terms in the flowlist are self-explanatory.

ØSLVL-(String search for level)- This subroutine is used to read one term of a string and to give a relative classification of the level of the term. A term is defined as a set of characters consisting of only the alpha characters and the star operator, or any valid set of characters which are enclosed in a set of parentheses or brackets. The level of a term indicates its ranking. Several terms that are connected by the AND, the OR, or the concatenation operators are on the same level. If these terms are enclosed in parentheses a new term is formed which has a

higher level than the individual terms. $SLVL gives a relative ranking of the level by the way the subroutine returns to the calling program. A standard return[9] is executed if terms of the same level follow. A non-standard return is executed if the term is followed by either a right parenthesis, right bracket, or an end flag; i.e., no terms of the same level follow.

The search for a term is started at the location given by RMP/RHP. This location should be on or before the start of the term. Upon return $SLVL contains the following information: the location of the start of the term is stored in LMP/LHP, the end of the term is given by LEMP/LEHP, the location of the character following the term is given by LRMP/LRHP, and "A" contains the character following the term.

Figure D.1 shows several examples of strings and the resulting locations. These locations are indicated by an arrow. The value of RMP/RHP before $SLVL is called, is represented by an S; all of the other locations given are those obtained after a return from $SLVL.

---

9 A standard return is a return to the location following the call statement. A non-standard return is a return to the second location following the call statement.

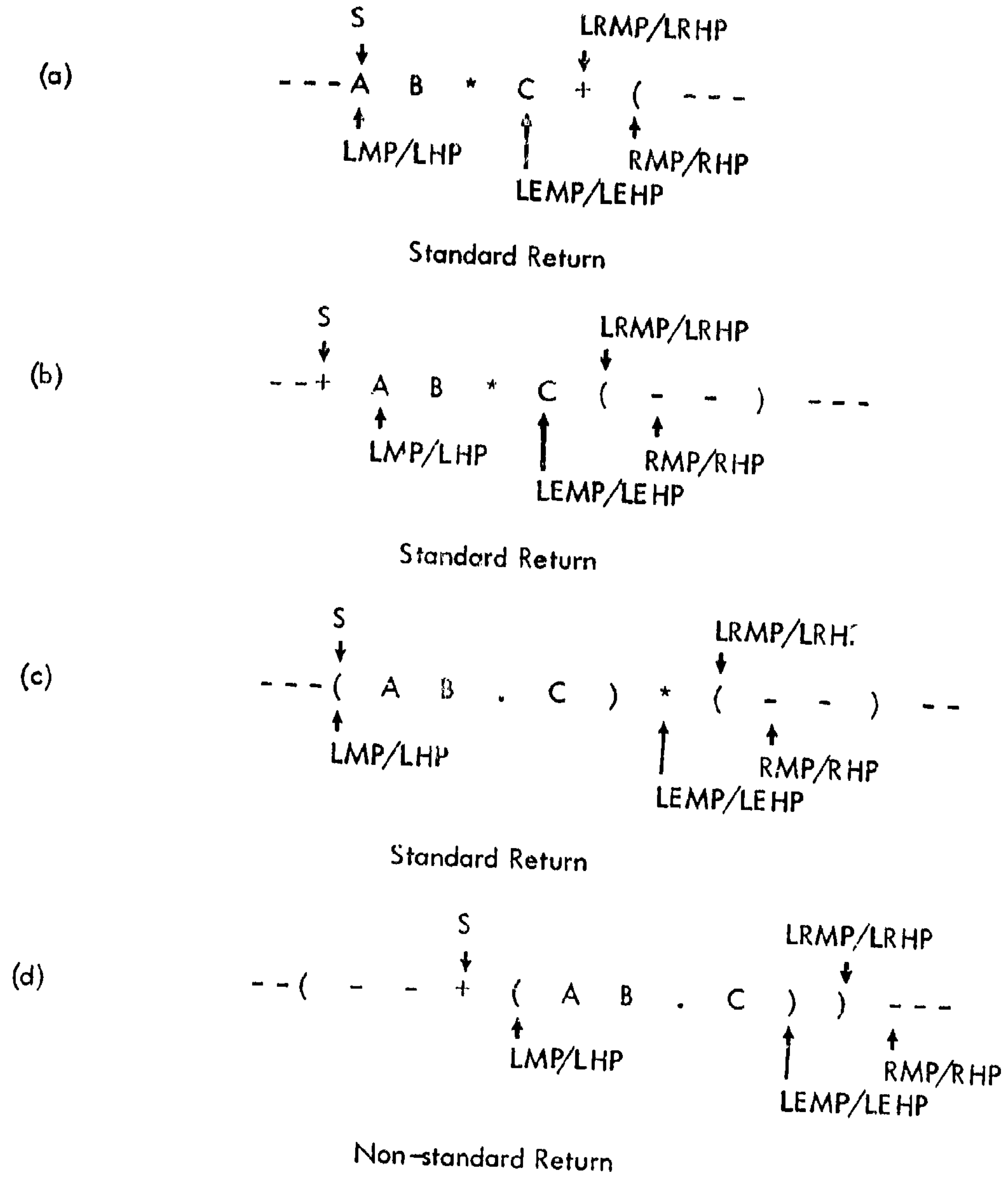


Figure D.1 The Operation of ØSLVL.

ØSLVL is often used to read all of the terms, in a string, which are on the same level. The set of operations needed to perform this is shown below.

→ Call: ØSLVL	
Jump	Standard return
Jump → $\underline{A}$	Non-standard return
RMP ← LRMP ←	Backspace read
RHP ← LRHP	
Jump	Read next term
$\underline{A}$	Continue

An explanation is needed for the use of the read location backspacing. Examining Figure D.1a it is seen that RMP/RHP gives the location of the first character of the following term so that ØSLVL can be called without modifying the read address. However, in the example shown in Figure D.1c. RMP/RHP gives the location of the second character in the following term, but LRMP/LRHP gives the location of the first term. Thus the read address is backspaced before calling ØSLVL a second time.

An outline of the operation of ØSLVL is given in Flowlist D.1.

FLOWLIST D.1 ﾘSLVL

```
    Enter: ØSLVL

    PC ← 0                                      Clear counter

┌──→Read one cell
│                                yes
│   IS: cell an alpha char.       → A
no
└── IS: cell = '(' or '['

    PC ← 1                                      Term enclosed in
                                                parenthesis

    LMP ← LRMP                                  Term start loc.

    LHP ← LRHP

┌──→Read one cell
│                                  no
│   IS: cell = '(' or '[' ────────────┐
│                                     │
│   PC ← PC + 1                       │
│                                     │
├── Jump                              │
no                                    │
├── IS: cell = ')' or ']' ←───────────┘

│   PC ← PC - 1
no
└── IS: PC = 0                                  Check for end of term

    LEMP ← LRMP ←─────────────┐                 Possible end of term
                              │
    LEHP ← LEHP               │
                              │
    Read next non-NULL char.  │                 Check for following '*'
                        yes   │
    IS: char. = '*' ──────────┘

    Jump → B
```

LMP ← LRMP ← <u>A</u>	Term consists of alpha char.
LHP ← LRHP	
LEMP ← LRMP	Assume end of term
LEHP ← LRHP	
Read next non-NULL char.	
IS: char = alpha or '*' — yes	
Continue ← <u>B</u>	End of term has been found and following character has been read
no — IS: char = ')' or ']' or '@ '	
Non-standard return	No similar level follows
Standard return	

$BPSET-(String break point set)- This routine is used to set a break point in a string. The purpose of the break point is used to insert characters in a string.

The break point is set after the last character which was read before calling $BPSET. This character appears in the accumulator and may be changed by modifying "A" before the call. The index register contains the flag, if any, and may not be changed. The first operation of $BPSET consists of rewriting this character in the string. The following 1, 2, or 3 characters are read and stored in SA1 and SA2. A flag is added to either SA1 or SA2 for use with $RTBP to set the return breakpoint. SRP is used to store the string return point. This is an address indicating where the string is to be continued.

After the characters have been stored a jump to the location given by NEW is added. Figure D.2 shows the memory, before and after the call, for several different types of strings. In these examples the break is to be made after the character 'A' and NEW is equal to '6000.

After $BPSET is called the characters to be inserted in the string are stored starting at the location given by NEW/0. This can be accomplished by using $COPY or $NWRT. $RTBP is then called to insert the characters which were removed by $BPSET and then sets a return jump to the location given by SRP.

An outline of the operation $BPSET is shown in Flowlist D.2.

Before '5000

A	B
C	D

After '5000

A	F-N
'6000	

SA1 = B
SA2 = C, F-d
SRP = '5002

(a)

Before '5000

	A
B	C

After '5000

	F-A
'6000	

SA1 = NULL
SA2 = B, F-C
SRP = '5002

(b)

Before '5000

A	F-B
'5500	

After '5000

A	F-N
'6000	

SA1 = F-B
SA2 = 0
SRP = '5500

(c)

Figure D.2 Examples of the Operation of $PBSET.

FLOWLIST D.2 $BPSET

Enter: $BPSET

APLUS ← "A"

XPLUS ← "X"

SA2 ← 0

IS: LRHP = 0 yes

"A" ← APLUS + flag

Store in cell at loc. given by LRMP/LRHP

Character last read is replaced with modified value and break point is set in 2nd cell

APLUS ← NULL

Jump → A

"A" ← APLUS

Store in cell at loc. given by LRMP/LRHP

Character last read is replaced with modified value in 1st cell

Read 2nd cell

APLUS ← "A"

XPLUS ← "X"

Store F-NULL at loc. given by LRMP/LRHP

This location is in 2nd cell

yes

IS: XPLUS = '40 ← A

Did last character read have flag

Jump → C

SA1 ← APLUS + XPLUS

SA1 contains only character + flag removed from string

LRMP ← LRMP + 1

RMP contains return loc.

Jump → B

NEW is stored in this loc.

Continue ← <u>C</u>	
SA1 ← APLUS	First Character or NULL removed from Ø
Read one character	
Exchange halves of "A"	
APLUS ← "A"	
Read next character	
"A" ← "A" + APLUS	"A" now contains both characters, less flag, stored in loc. following break point. "X" contains flag.
IS: "A" = NULL, NULL yes	Are both characters NULL's
SA2 ← "A" + flag	Set return flag
Jump → <u>B</u>	
SA1 ← SA1 + flag	Set return flag
Store NEW at (LRMP)* ← <u>B</u>	Set break address
SRP ← RMP	Set return address
Return	

$RTBP-(string return breakpoint)-This subroutine is used to return the breakpoint which was set by $BPSET. $RTBP adds the character which was stored in SA1 and then adds the two characters stored in SA2. The return address is stored in the string and the value of NEW is updated. The read address is set to the string location containing the character which was passed via SA1.

It should be noted that before calling $RTBP, WMP/WHP contains the location of the first cell following the inserted string. This address resulted from the routines which inserted the string; $COPY, $WRITE, or $NWRT. WMP/WHP may not be modified before the call to $RTBP.

$RTBP checks the data that was passed via SA1 and SA2 to see if they contain any information. SA1 contains only one character and this character will contain information only if it does not contain a NULL without a flag. If SA2 contains any information it will contain at least a flag, otherwise, SA2 will be zero and a flag is added to SA1.

The reader should refer to Flowlist D.3 fc outline of the operation of this routine.

FLOWLIST D.3 ØRTBP

Enter: ØRTBP	
RMP ← WMP	Set read loc. to 1st string add loc.
RHP ← WHP	
IS: WHP = 0 (no →)	Check loc. of next cell
IS: SA1 = '36 (yes) → A	Next cell is in 1st half of word. Check SA1 for NULL-without a flag
Write NULL in 1st cell	SA1 is to be written. Write NULL to fill 1st cell
Write SA1 in 2nd cell ←	Write SA1 in 2nd, even if it does not contain information, to fill out word
(no) IS: SA2 = 0 ← A	
Store SA2 at (WMP)*	SA2 contains information. Store both characters at once
WMP ← WMP + 1	
→ Store SRP at (WMP)*	Store return loc.
NEW ← WMP + 1	Update NEW
Return	

ØDCLAS -(string derivative classification)-This subroutine is called by DCONT to classify the substrings, of the work string, as given below. After classifying ØDCLAS calls the appropriate subroutine which does the actual expansion. The different forms of the substrings are listed below with their expanded form on the right.

D.1 $\quad$ D ((Ø)*) = D (Ø) (Ø)*

D.2a $\quad$ D ((Ø1) (Ø2)) = (D (Ø1) (Ø2) + D (Ø2)), η (Ø1) = λ

D.2b $\quad$ D ((Ø1) (Ø2)) = D (Ø1) (Ø2), η (Ø1) = Ø

D.3 $\quad$ D (Ø1 + Ø2) = (D (Ø1) + D (Ø2))

D.4 $\quad$ D ([Ø]) = [D (Ø)]

D.5 $\quad$ *

D.6 $\quad$ D ((Ø)) = D (Ø)

D.7a $\quad$ D (A*. . . .) = D ((A)*. . . .)

or

D.7b $\quad$ D (A. . . . .) = D (A. . . .)

D.8 $\quad$ D (Ø1 . Ø2) = (D (Ø1) . D (Ø2)), where 'A' is one or more literals and 'Ø' represents any substring.

ØDCLAS starts at the left of the string searches for the 'D' and saves its location in THP/TMP. The following '(' is then found and its location stored in T2HP/T2MP.

This parenthesis forms the start of a level. The number and type of the terms

* The first eight forms are numbered according to the subroutines which use these forms. Due to a reorganization of the subroutines there is no form D.5.

is tested to find the characteristics of the substring. Figure D.3 shows the different characteristics for the different forms and also the set of pointers which gives the location of parts of the substring. For substrings of the forms 2 or 7 the final testing needed to indicate the proper expansion is contained in the individual expansion routines. It should be noted that forms 3 and 8 are identical except for the connective operator and are expanded by the same subroutine.

After the substring is classified the proper expansion subroutine is called. These subroutines are titled according to the form they handle; e.g., $DF1 expands substring of form 1. If any changes were made in the string by the expansion subroutines this is indicated by incrementing DPC. When these subroutines return to $DCLAS the read address (RHP/RMP) is left pointing to one of the characters in that portion of the string which was modified. $DCLAS then searches for the next character 'D' and the process is repeated.

$DCLAS returns to the calling routine when it encounters the end flag. The calling routine then tests to see if any changes were made in the string. If changes were made then $DCLAS is called again to see if any new forms were generated during the last expansion. This process is repeated until there is a pass with no changes.

The reader is referred to Flowlist D.4 for an outline of this routine.

1	2	3/8	4	6	7	FORM OF SUBSTRING	
1	2 or more	2 or more	1	1	1	Number of terms	CHARACTERISTICS
	Juxta-position	AND/OR				Connective Operator	
*			]	)		Last character of term	
[, (						First character of term	
D	D	D	D	D	D	THP/TMP	POINTERS
1st (	1st (	1st (	1st (	1st (	1st (	T2HP/T2MP	
*			]	)		LEHP/LEMP	
	First character of 1st term					LHP/LMP	
	Last)					LRHP/LRMP	

Figure D.3 Characteristics of Substrings.

FLOWLIST D.4 ØDCLAS

Enter: ØSCLAS	
RHP/RMP ← 0/DRSTRT	Set string starting address
Read a character ← A	
no IS: Char. = '@'	
Return	End of one pass
IS: Char. = 'D' no → A	
THP/TMP ← LRHP/LRMP	Save loc. of 'D'
Find '('	
Save loc. in T2HP/T2MP	
Read a term	ØSLVL
IS: Term last in level yes → B	
no IS: Term followed by '+' or '.'	Forms 2, 3 or 8
Call: ØDF38	Form 3 or 8
Jump → A	
Read next term	
no IS: Term last in level	
Jump	
Call: ØDF2	Form 2
Jump → A	

Continued on following page

Continued ← B

no

IS: Last char. = ')' — Loc. of end character given by LEHP/LEMP

Call: ØDF6 — Form 6

Jump → A

no

IS: Last char. = ']'

Call: ØDF4 — Form 4

Jump → A

yes

IS: Last char. = '*'

Call: ØDF7 — Form 7

Jump → A

no

IS: First Char. = '(' — Loc. of first character given by LHP/LMP

Call: ØDF1 — Form 1

Jump → A

ØDF1-(string expansion form 1)-This subroutine is called by ØDCLAS to expand strings of the form 'D ((Ø)*)' to strings of the form 'D (Ø) (Ø)*'.

An outline of this routine is shown in Flowlist D.5. The reader should note that this routine (and some of the other expansion routines) uses the end flag ('@ ') not only to indicate the end of a string, but that it is also used to indicate the end of the substring that is being tested or copied. In the flowlist the comments column is used to represent the string during the different parts of the expansion process.

FLOWLIST D.5 ØDF1

Enter: ØDF1	
DPC ← DPC + 1	A change is made in string
RHP/RMP ← LHP/LMP	
Read term	Via ØSLVL
Replace following char. with '@ '	D ((Ø)*@
Set breakpoint	Via ØBPSET
RHP/RMP ← LHP/LMP	Start of term
Copy term to flag	D ((Ø)*@ (Ø)*
Return breakpoint	Via ØRTBP
RHP/RMP ← LEHP/LEMP	End of term
Read a character	Reads '*'
Replace with NULL	D ((Ø) @ (Ø)*
Find flag	
Replace with ')'	D ((Ø)) (Ø)*
Return	

ØDF2-(string expansion form2)-This subroutine is called by ØDCLAS to perform the expansion given below.

$$D\left((\text{Ø}1)(\text{Ø}2)\right) = \begin{cases} D(\text{Ø}1)(\text{Ø}2), & \eta(\text{Ø}1) = \text{Ø} \\ (D(\text{Ø}1)(\text{Ø}2) + D(\text{Ø}2), & \eta(\text{Ø}1) = \lambda \end{cases}$$

It should be remembered from the description of ØDCLAS that LRHP/LRMP points to the last right parenthesis in this substring. This fact is used to set a flag which indicates the end of the second term. A flag is also placed at the end of the first term to indicate its end. Thus when ØETATT is called the start of the first term is given by LHP/LMP and it ends when the end flag is found.

ØETATT is the subroutine used to test a substring to see if it contains lambda; i.e., it tests if η (Ø) = λ. The result is returned via the variable ETAVAL. ETAVAL is set to one if the substring contains lambda; otherwise, it is set to zero.

The expansion of the form 2 substring is shown in Flowlist D.6.

FLOWLIST D.6 ØDF2

Enter: ØDF2	
DPC ← DPC + 1	
Replace last ')' with ' @ '	D ((Ø1) (Ø2) @
RHP/RMP ← T2HP/T2MP	
Read first '('	
Read following term	
Store following character in XTEMP	
Store location in T2HP/T2MP	
Replace with ' @ '	D ((Ø1) @ Ø2) @
Test η (Ø1)	Via ØETATT
IS: ETAVAL = 0 $\xrightarrow{\text{no}}$ B	
RHP/RMP ← T2HP/T2MP	Does not contain lambda
Replace flag with ')'	D ((Ø1) (Ø2) @
Set breakpoint	
Add character stored in XTEMP	D ((Ø1)) (Ø2) @
Return breakpoint	
Find end flag	
Replace with NULL	D ((Ø1)) (Ø2)
Return	

Continued on following page.

Continue ← B	Contains lambda
RHP/RMP ← T2HP/T2MP	
Replace flag with ')'	
Set breakpoint	
Add character stored in XTEMP	D ((Ø1)) (Ø2)@
Return breakpoint	
Find end flag	
Set breakpoint	
Add 'D('	D ((Ø1)) (Ø2) @ D(
RHP/RMP ← T2HP/T2MP	Start of second term
Copy second term to flag	Via ØCOPY D ((Ø1)) (Ø2) @ D ((Ø2)
Add '))'	D ((Ø1)) (Ø2) @ D ((Ø)))
Return breakpoint	
Find flag	
Replace with '+'	D ((Ø1)) (Ø2) + D ((Ø2)))
RHP/RMP ← THP/TMP	Address of first 'D'
Read character	
Replace with '('	
Set breakpoint	
Add 'D'	(D ((Ø1)) (Ø2) + D ((Ø2)))
Return breakpoint	
Return	

ØDF38-(string expansion forms 3 and 8)-ØDF38 is called by ØDCLAS to expand the strings given below.

D (Ø1 + Ø2) = (D (Ø1) + D (Ø2))

D (Ø1 . Ø2) = (D (Ø1) . D (Ø2))

The expansion of these two forms are identical except for the operator which connects the two terms, thus ØDF38 can perform both expansions. In fact ØDF38 does not test for the operator, but finds it and stores it until needed without determining the operator. The example string shown in Flowlist D.7 arbitrarily uses the operator '+'.

The reader should note that in this expansion (and in some of the other expansion forms) that the string is expanded from right to left. This procedure is dictated by the subroutine ØBPSET which is used to see the string breakpoint. When ØBPSET is called several characters in the cells, following the location where the breakpoint is set, are moved. Thus, this character will no longer be stored in the cells whose locations were given by THP/TMP, T2HP/T2MP, etc.

FLOWLIST D.7 ØDF38

Enter: ØDF38	D (Ø1 + Ø2)
DPC ← DPC + 1	
RHP/RMP ← T2HP/T2MP	Location of first '('
Read term	Finds end ')'
RHP/RMP ← LEHP/LEMP	Location of end ')'
Read a character	
Set breakpoint	
Add ')'	
Return breakpoint	D (Ø1 + Ø2))
RHP/RMP ← T2HP/T2MP	
Read one character	Reads '('
Read following term	Find '+' or '.'
no	
IS: Following character = '+' or '.'	
Save character in XTEMP	
Set breakpoint	
Replace operator with ')'	D (Ø1)Ø2))
Add character stored in XTEMP	D (Ø1) + Ø2))
Add 'D('	D (Ø1) + D (Ø2))
Return breakpoint	

RHP/RMP ← THP/TMP	
Read a character	Reads 'D'
Set breakpoint	
Replace character with '('	((Ø1) + D (Ø2))
Add 'D'	
Return breakpoint	(D (Ø1) + D (Ø2))
Return	

$\not{S}$DF4-(string expansion form 4)-$\not{S}$DF4 is called by $\not{S}$DCLAS to expand the substring given below to the form on the right.

$$D\,([\not{S}]) \;=\; [\,D\,(\not{S})\,]$$

The operation of this expansion subroutine is given in Flowlist D.8

This routine is based on the use of $\not{S}$SLVL and thus gives the reader a chance to study the operation of $\not{S}$SLVL without a lot of other operations being performed. When $\not{S}$DF4 is called the location of the start of the inner term (the '[$\not{S}$]') is given by LHP/LMP. This address is transferred to the read address (RHP/RMP) and the term is read via $\not{S}$SLVL. $\not{S}$SLVL returns with the address of the first character (the '[') in LHP/LMP, the address of the last character (the ']') in LEHP/LEMP, and the address of the following character (the ')') in LRHP/LRMP.

FLOWLIST D.8 ∅DF4

Enter: ∅DF4	D ([∅])
DPC ← DPC + 1	
RHP/RMP ← LHP/LMP	
Read one term	Via ∅SLVL
Replace following character with ']'	D ([∅]]
RHP/RMP ← LEHP/LEMP	
Read character	Reads ']'
Replace with ')'	D ([∅)]
RHP/RMP ← LHP/LMP	
Read character	Reads '['
Replace with '('	D ((∅)]
RHP/RMP ← T2HP/T2MP	
Read character	Reads '('
Replace with 'D'	DD (∅)]
RHP/RMP ← THP/TMP	
Read character	Reads 'D'
Replace with '['	[D (∅)]
Return	

ØDF6-(string expansion form 6)-This subroutine is used to expand the substring given below to the form on the right.

$$D\ ((\emptyset))\ =\ D\ (\emptyset)$$

When ØDF6 is called the location of the inner left parenthesis is given by LHP/LMP, while the location of the inner right parenthesis is given by LEHP/LEMP. This routine replaces the inner parenthesis with NULL's.

ØDF7-(string expansion form 7)-This routine is used to expand the form given below.

D (A*. . . .A) = D ((A)*. . . .A), where A is any member of the set of alpha characters (the literals, lambda, and phi). If this form does not exist, no change is made in the string. The purpose of this expansion is to prepare the string for processing by ØDF2 and ØDF1 on the following passes.

During the last pass through ØDCLAS all of the substrings are of the form 'D (A. . . .A)' thus only ØDF7 is called to expand the substrings. As ØDF7 makes no change in these substrings DPC is not incremented and its value remains zero. A DPC of zero is used to indicate to DCONT that the last pass has been made.

The operation of ØDF7 is shown in Flowlist D.9.

FLOWLIST D.9 ØDF7

Enter: ØDF7	
RHP/RMP ← LHP/LMP	Term start
Read character	
no	
IS: It an alpha character	
Store location at LHP/LMP	
Jump	
yes	
IS: Character = '*'	
Return	Make no changes
DPC ← DCP + 1	D (A*. . .A)
RHP/RMP ← LHP/LMP	
Read character	Reads alpha character
Store character in XTEMP	
Set breakpoint	
Replace character with '('	D ((*. . .A)
Add character stored in XTEMP	D ((A*. . .A)
Add ')'	D ((A)*. . .A)
Return breakpoint	
Return	

S̸ETATT-(string eta function test)-S̸ETATT is called by S̸DF2 to calculate the eta function of a substring. The starting address of this string is passed in LHP/LMP and the end is indicated by the end flag. The eta function is used to determine if a string contains lambda, thus S̸ETATT is also called by DCONT to test the final string; giving the operator the information needed to determine the output of the state associated with this string.

The following equation defines the eta function.

$$\eta\,(\not S) = \begin{cases} \lambda, & \text{if } \lambda \in \not S \\ \emptyset, & \text{if } \lambda \notin \not S \end{cases}$$

From this definition the following rules can be obtained.

D.9 $\quad \eta\,(L) = \emptyset$, where L is any literal

D.10 $\quad \eta\,(\lambda) = \lambda$

D.11 $\quad \eta\,(\emptyset) = \emptyset$

D.12 $\quad \eta\,(\not S^*) = \lambda$, where $\not S$ is any substring

D.13 $\quad \eta\,((\not S1)\,(\not S2)) = \eta\,(\not S1)\,.\,\eta\,(\not S2)$

D.14 $\quad \eta\,(f\,(\not S1, \not S2)) = f\,(\eta\,(\not S1), \eta\,(\not S2))$, where f is any Boolean function.

From these rules we can see that the eta function can be calculated as a Boolean function if the concatenation operator is treated as the Boolean AND operator. This procedure was used in the design of S̸ETATT. The resulting value of the eta function was encoded as the Boolean variable ETAVAL with a one representing the condition where $\eta = \lambda$.

The subroutine S̸ETATT is used to implement rules 5 and 6, while S̸BETAT is

called to implement the first four rules. To understand the logic of ØETATT it is necessary to introduce the following Boolean identities.

D.15 $1 + X = 1$

D.16 $0 + X = X$

D.17 $1 \, . \, X = X$

D.18 $0 \, . \, X = 0$, where X is either 1 or 0.

These identities are used to determine if a given term of the substring needs to be tested and how the results are combined. To perform this determination ØETATT was broken down into four modes or sub-sections.

<u>AND-Lambda, OR-Phi Mode</u>. This mode is used to implement rules D.16 and D.17. These rules state that if the previous term had a value of one and was followed by the AND operator (or concatenation) or if the previous term had a value of zero and followed by the OR operator then the results are determined solely by the present term. When ØETATT is called the results are determined solely by the term to be tested; thus ØETATT starts in this mode. In this mode the term is read by ØSLVL to obtain the starting and end addresses needed by ØBETAT.

After ØBETAT is called it can return one of three values. If ØBETAT executes a standard return then ETAVAL will either be one or zero depending on whether the term tested contained, or did not contain, lambda. Under this condition ØETATT goes to the NEXT mode which determines the next mode of the subroutine. On the other hand, a non-standard return is executed by ØBETAT to indicate that the term was too complicated to evaluate. To evaluate this term it is necessary to simplify the term by "going down in level".

OR-Lambda Mode. This mode is used to implement equation D.15. This equation states that the results are known and that there is no need to test the following terms on the same level. In this mode each term is read until the end of the level is reached, at which time control goes to the NEXT mode.

AND-Phi Mode. In this mode, an implementation of equation D.18, all terms are read and passed over until either the end of level is reached, or the OR operator is found. When the end of the level is reached control goes to the NEXT mode. If the OR operator is found the control will go to the OR-Phi mode.

NEXT Mode. This mode is used to test for the end of a string, find the next mode, and to implement the operation of negation. When this section is entered the character following the last level has been read and it is stored in the accumulator for testing.

If this character is the end flag then the end of the substring has been found and ØETATT returns with the current value of ETAVAL. If a right bracket is found (this implies that the corresponding left bracket was passed over in the process of "going down in level") the present value of ETAVAL is negated. The finding of the OR operator causes control to go to the OR-Phi or the OR-Lambda mode depending on whether ETAVAL equals zero or one. The finding of the AND operator, or the concatenation operator, causes control to go to either an AND-Phi or the AND-Lambda mode depending on the value ETAVAL. The finding of either the OR, AND, or concatenation operators indicate that a new term follows. Thus control is passed to the appropriate mode to evaluate this term.

If control has not yet branched to another mode then the character (by

elimination we can see that the character is either the ')' or the ']') is ignored and the next character is read. Control remains in the NEXT mode and this process is repeated.

The operation of ∅ETATT is shown in Flowlist D.10.

FLOWLIST D.10 ØETATT

Enter: ØETATT	
RHP/RMP ← LHP/LMP	
Read a term ← A	AND-Lambda, OR-Phi Mode. Find address for ØBETAT
Call: ØBETAT	
IS: Return standard (yes)	
RHP/RMP ← LHP/LMP	No decision
Read first character of term	Go down in level
Jump → A	
RHP/RMP ← LHP/LMP	Finds character following term
Read the term	
Jump → B	Find next mode
Read a term ← C	OR-Lambda Mode
IS: Term last in level (no)	
Jump → B	Find next mode
Read a term ← D	AND-Phi Mode
IS: Term last in level (yes) → B	Find next mode
IS: Term followed by '+' (no)	
Jump → A	OR-Phi Mode

Continued on following page.

Continue ← B	Finds next mode
no	
IS: Term followed by '@ '	
Return	Done
no	
IS: Term followed by ']'	
ETAVAL ← ~ ETAVAL	
no	
IS: Term followed by '+'	
yes	
IS: ETAVAL = 1 → C	OR-Lambda mode
Jump → A	OR-Phi mode
yes	
IS: Term followed by Concatenation	A '(', '[', or alpha character
yes	
IS: Term followed by '.'	
Read next character	
Jump → B	
Continue	
yes	
IS: ETAVAL = 1 → A	AND-Lambda mode
Jump → D	AND-Phi mode

ØBETAT-(Basic eta function)-This subroutine is called by ØETATT to implement the first five rules of the eta function (as given in the previous section of ØETATT). The start of the term to be tested is passed via LHP/LMP while the end of the term is given by LEHP/LEMP.

If the term is enclosed in parentheses (or brackets) it will contain lambda if the star operator follows the term (Rule D.12). This is indicated by executing a standard return with ETAVAL equal to one. On the other hand, if the term is not followed by the star operator then lower level term(s) which form this term have to be tested to determine the results. This is indicated by executing a non-standard return to ØETATT which finds these lower level term(s).

If the term is not enclosed in parentheses then it consists of only alpha characters and the star operator (as given by the definition of "term" in the section on the subroutine ØSLVL). From the first five rules of the eta function we can see the term contains lambda if, and only if, it consists of the following.

D.19 'λ'

D.20 'λ*'

D.21 'L*', where L is any one literal

D.22 'Ø*'

D.23 Or any of the above terms concatenated.

ØBETAT tests the terms for the above properties and executes a standard return when the end of the term is found. At this time ETAVAL will be equal to one if the term had the above properties, otherwise ETAVAL will be equal to zero.

Flowlist D.11 shows the operation of ØBETAT.

FLOWLIST D.11 ØBETAT

Enter: ØBETAT	
ETAVAL ← 0	
RHP/RMP ← LHP/LMP	Address of 1st character of term
Read character	
IS: Character = '(' or '[' $\xrightarrow{\text{yes}}$ B	
IS: Character an alpha chor. (yes) ← A	Term consists of alpha char. and '*'
Return	End of term
→ IS: Character = 'λ' (yes)	
Read next character	A literal or phi must be followed by a '*'
IS: Character = '*' (no)	
ETAVAL ← 1	
Read next character	
Jump → A	
→ ETAVAL ← 0	Does not contain λ
Return	
ETAVAL ← 1 ←	A 'λ' may, or may not be followed by '*'
Read next character	
IS: Character = '*' (no)	
Read next character	
→ Jump → A	

Continued on following page.

Continued $\leftarrow$ $\underline{A}$	
RHP/RMP $\leftarrow$ LEHP/LEMP	Term is enclosed in parentheses or brackets
Read character	Reads last character of term
no	
IS: Character = '*'	
ETAVAL $\leftarrow$ 0	Contains lambda
Return	
Non-standard return	No decision made

$MATCL-(string match and clear)-This subroutine is called by DCONT to perform the operation of taking the derivative and simplifying the string. The final derivative operation is defined as,

$$D/A/(A_1A_2 - - - A_N) = \begin{cases} \lambda(A_2 - - - A_N), & \text{if } A = A_1 \\ \emptyset, & \text{if } A \neq A_1 \end{cases}$$

$MATCL performs the above operation by comparing the derivative name character (passed via CHRMAT) with the first character in the term enclosed by the characters 'D(' and by ')'. During this operation these enclosing characters are removed.

When the end flag is found the operation halts and four subroutines ($1SIM, $2SIM, $3SIM, and $4SIM) are called to simplify the string. Simplification is an iterative process similar to the expansion process performed by $DCLAS. In other words multiple passes are made through the simplification routines until a pass is found during which no changes were made. The number of changes made are tallied in the variable DPC. It should be noted that the read address is set to point to the start of the string before calling, rather than within the subroutines.

The operation of $MATCL is shown in Flowlist D.12.

FLOWLIST D.12 $MATCL

Enter: $MATCL	
RHP/RMP ← 0/DRSTRT	Starting address
Read character ← <u>A</u>	
IS: Character = '@' (no → <u>C</u>)	
RHP/RMP ← 0/DRSTRT	
Simplify string	Via $1SIM, $2SIM, $3SIM, and $4SIM
IS: DPC = 0 (no → RHP/RMP ← 0/DRSTRT)	Any changes made?
Return	
IS: Character = 'D' → (no) <u>A</u>	Find next 'D'
Replace 'D' with NULL	
Read next character	Reads '('
Replace with NULL	
Read next character	Reads first alpha character
IS: Character = CHRMAT → (no) <u>B</u>	
Replace with 'λ'	A match
Find next ')'	
Replace with NULL	
Jump → <u>A</u>	Find next term

Continue ← B

Replace character with Ø — No match

Read next character

yes

IS: Character = ')'

Replace with NULL

Jump

Replace with NULL

Jump → A — Find next term